NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

P7444

DESIGN OF A SYNTAX DIRECTED
EDITOR FOR PSDL

by

Scott Wilkin Porter

December 1988

Thesis Advisor: Valdis Berzins

Approved for public release; distribution is unlimited.

T242269

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (*If applicable*) 52	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (*City, State, and ZIP Code*) Monterey, California 93943-5000	7b. ADDRESS (*City, State, and ZIP Code*) Monterey, California 93943-5000
8a. NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)

DESIGN OF A SYNTAX DIRECTED EDITOR FOR PSDL

12. PERSONAL AUTHOR(S)
Porter, Scott W.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Master's Thesis	FROM _____ TO _____	1988 December	145

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	Prototyping, Syntax Directed Editor, Editor Generator, Software Engineering

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

Computer Aided Prototyping System (CAPS) is a programming tool which uses PSDL (Prototype Systems Design Language) as a specification language for large ADA program development. The CAPS uses a syntax directed editor as a part of the user interface for the system. This thesis focuses on the specification and design of the syntax directed editor for PSDL using the Cornell Synthesizer Generator to create a language-based editor.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Valdis Berzins	22b. TELEPHONE (*Include Area Code*) (408) 646-2461 22c. OFFICE SYMBOL 52 Be

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

☆ U.S. Government Printing Office: 1986—606-24

Approved for public release; distribution is unlimited.

DESIGN OF A SYNTAX DIRECTED EDITOR FOR PSDL

by

Scott Wilkin Porter
Lieutenant, United States Navy
B.S., Auburn University, 1980

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1988

ABSTRACT

Computer Aided Prototyping System (CAPS) is a programming tool which uses PSDL (Prototype Systems Design Language) as a specification language for large ADA program development. The CAPS uses a syntax directed editor as a part of the user interface for the system. This thesis focuses on the specification and design of the syntax directed editor for PSDL using the Cornell Synthesizer Generator to create a language-based editor.

C.1

TABLE OF CONTENTS

I. INTRODUCTION.....1

A. OVERVIEW.....1

B. DESCRIPTION OF SOFTWARE ENGINEERING PRINCIPLES AND FOURTH GENERATION LANGUAGES.....1

1. Software Engineering.....1

2. Ada.....3

3. Prototype System Design Language (PSDL).....3

C. SYNTAX DIRECTED EDITOR GENERATORS.....4

1. Cornell Synthesizer Generator.....5

2. GANDALF.....11

a. Project Management SDC (System Development Control System).....12

b. System Version Control (SVCE).....13

c. Incremental Programming LOIPE (Language Oriented Incremental Programming Environment).....14

d. Syntax Directed Editing ALOE (A Language Oriented Editor).....14

e. The GANDALF Prototype.....17

f. Designing the Environment.....18

g. An Example of an ALOE.....18

h. Generating an ALOE.....19

i. Programming Environments.....22

j. User Interface.....22

3. Evaluation and Comparison of the Cornell Synthesizer and GANDALF.....22

II. A DESCRIPTION OF THE PROTOTYPE SYSTEM DESIGN LANGUAGE (PSDL).....25

A. PROTOTYPE SYSTEM DESIGN LANGUAGE (PSDL).....25

1. Model.....26

2. Operators.....27

3. Communicators.....27

4. Abstractions.....28

5. Timing.....31

6. Hierarchy.....31

B. ATTRIBUTE GRAMMARS.....32

C. OVERVIEW OF COMPUTER AIDED PROTOTYPING SYSTEM....34

D. BACKUS-NAUR FORM FOR PSDL.....36

III. THEORY OF CORNELL SYNTHESIZER GENERATOR.....37

A. TRANSLATOR AND KERNEL.....37

B. BUFFERS.....37

C. TRANSFORMATIONS.....38

D. EDITING.....39

E. ATTRIBUTION.....40

F. FILE REPRESENTATION.....40

G. PRACTICAL MATTERS CONCERNING IMPLEMENTATION.....40

IV. CONCEPTUAL DESIGN AND SOFTWARE ENGINEERING OF A PSDL SYNTAX DIRECTED EDITOR.....45

A. REQUIREMENTS ANALYSIS.....45

1. Operating Environment.....45

2. System Goals.....45

3. Performance Constraints.....46

4. Implementation Constraints....46
5. Resource Constraints....47
6. External Interfaces....47
B. FUNCTIONAL SPECIFICATIONS....48
C. ARCHITECTURAL DESIGN....49
D. IMPLEMENTATION....50
1. Abstract Syntax Specification....51
2. Attributes and Attribute Equations....54
3. Unparsing Declarations....57
4. Concrete Input Syntax....59
5. Transformations and Templates....62
E. EVOLUTION....63
V. USER GUIDE TO THE PROTOTYPING LANGUAGE EDITOR....64
A. GENERIC FEATURES OF THE EDITOR....64
B. LANGUAGE SPECIFIC FEATURES OF THE EDITOR....69
C. USING THE EDITOR....69
1. Structural Editing....70
2. Textual Editing....72
3. Demonstration of Prototyping Language Editor....72
VI. CONCLUSIONS AND FOLLOW-ON WORK....77
APPENDIX A: BACKUS-NAUR FORM FOR PSDL....79
APPENDIX B: COMPLETE PSDL ABSTRACT SYNTAX....83
APPENDIX C: PSDL.SSL CURRENT PARTIAL IMPLEMENTATION....92
APPENDIX D: UNPARSING DECLARATIONS....94
APPENDIX E: CONCRETE INPUT SYNTAX....100

APPENDIX F: PROTOTYPING LANGUAGE EDITOR COMMANDS.....102

APPENDIX G: PARTIAL GLOSSARY OF TERMS FROM THE SYNTHESIZER GENERATOR REFERENCE MANUAL.....112

LIST OF REFERENCES.....130

INITIAL DISTRIBUTION LIST.....132

LIST OF FIGURES

1. Syntax Directed Editor Generator.....4
2. Cornell Synthesizer Generator.....6
3. Derivation Tree.....7
4. A Language Oriented Editor (ALOE).....16
5. Grammar for Pascal-like Variable Declarations.....20
6. Abstract Syntax Description.....21
7. PSDL Computational Model.....26
8. Derivation Tree.....33
9. Abbreviated PSDL in Backus-Naur Form.....41
10. SSL Modules in psdl.syn.....50
11. Abstract Syntax for Abbreviated PSDL Grammar.....55
12. Grammar Rules Defining Display.....58
13. Association Between Abstract and Input Syntax.....60
14. Translation of Input Syntax into Abstract Syntax.....61
15. Regular Expression Notation.....61
16. Syntax of Concrete Input.....62
17. Editing Session Screen.....64
18. Mouse Commands.....67

ACKNOWLEDGEMENT

I would like to thank Professor Valdis Berzins for his help and guidance. His willingness to take on a thesis student late in the cycle whose thesis topic was somewhat removed from his own research efforts is a testimony to his dedication, professionalism, and sincere concern for his students.

I would also like to thank Professor Luqi without whom this thesis would have never been undertaken. I am especially grateful for the guidance and direction she provided to the CAPS thesis students as a group.

Any serious attempt to get the school computers to cooperate or do anything productive would have been futile without the capable assistance of Rosalie Johnson and Susan Whalen.

I owe a special debt to Al Wong and Lcdr John Yurchak for having the patience and perseverance to drum in the information necessary to get me started with phyla, trees, productions and all manner of strange creatures.

It was a pleasure to work with such a capable group of CAPS co-conspirators. The unity, cooperation and concern kept the glimmer at the end of the tunnel shining even when the facts didn't support such optimism.

A special thank you to my wife and thesis manager Drue and children Scotty, Maggie, and Katie, not only for their patience, love and prayers, but also for supporting me in every area to complete my thesis.

I. INTRODUCTION

A. OVERVIEW

Prototype System Design Language (PSDL) is the essence of the user interface in the Computer Aided Prototyping System (CAPS). It provides the CAPS user with a method of specifying very complex ideas simply and concisely. PSDL, coupled with the CAPS system, allows the generation of prototypes that are models of the final product as conceived by the programmer and partial construction of the final project. Early prototyping allows specification refinement and shows the customer what the proposed system will do prior to investing large amounts of money in deficient directions. The benefits of this system of software development include increased productivity, the ability to undertake larger projects, improved reliability, better cost estimation, reduced development costs, the ability to perform feasibility studies.

B. DESCRIPTION OF SOFTWARE ENGINEERING PRINCIPLES AND FOURTH GENERATION LANGUAGES

1. Software Engineering

Software engineering is the application of engineering disciplines and mathematical principles to the problem of creating software that efficiently and correctly solves problems on computer hardware. Software consists of

the auxiliary packages required to perform specific applications on hardware and includes programs, user guides, documentation, validation, and any other special information necessary to correctly operate them. One of the primary goals of software engineering is to ascertain precisely what the customer or end user wants and to be able to communicate about that with him. This contributes to increased efficiency which is itself another major goal of software engineering, that is, the ability to turn out the correct program at minimal cost both in terms of time and money. In keeping with the reduction of costs, the introduction of standard methodologies based on sound theoretical principles increases consistency which also improves the reusability and readability of the product. Another fundamental goal is accuracy, the lack of which can lead to high costs in lives, equipment, and money. [Ref. 1]

One of the fundamental keys to the advancement of computer science and software engineering is the application of abstraction. Abstraction is the reduction of a system to only the details that are important for a particular purpose. The result of abstraction is a reduction of the quantity of details required to understand the system in question, allowing the comprehension by humans of larger and more significant projects. A goal in software engineering is to increase the level of automation involved in the development of software. In order to automate, the language used must be

precisely defined in terms of syntax and semantics. In the CAPS system, two such languages have been chosen, PSDL and Ada to partially automate the software engineering task.

2. Ada

Ada, a fourth generation language, was developed by the United States Department of Defense (DoD) primarily to standardize the programming language for military applications. Of particular concern were embedded or mission critical computer applications. Included in the specifications for this language were readability, documentation, simplicity, modularity supporting information hiding, concurrency, and amenability to correctness verification. The improvements in fourth generation languages include the linguistic support for information hiding and concurrent programming by providing an encapsulation facility supporting the isolation of specification and definition, information hiding, name access by mutual consent and generic modules. [Ref. 2]

3. Prototype System Design Language (PSDL)

Prototype System Design Language (PSDL) is a high level prototyping language that can specify a program in sufficient detail to be able to identify reusable software from an on-line database and iteratively refine and further define the prototype program. In the context of the Computer Aided Prototyping System, the idea is that the designer and the customer should be able to quickly determine the

specifications of the proposed system, after which the Computer Aided Prototyping System searches for appropriate reusable pieces of software from the software database.

PSDL's features include definition of real-time timing constraints and the ability to design in a top-down fashion by decomposing higher level entities into lower level components. PSDL creates an executable prototype. The PSDL computational model specifically deals with the interaction between components and makes modularization easier to implement. [Ref. 3]

C. SYNTAX DIRECTED EDITOR GENERATORS

Syntax directed editors are editors tailored to a specific language. Using the grammar, format, and semantics of a language, they assist in the writing of correct programs by disallowing incorrect code and by prompting or inserting correct program segments and legal alternatives. Since the core of most editors is very similar, syntax directed editor generators have been developed to capitalize on the similarities and only require specification of language specific details (see Figure 1). The predominant syntax

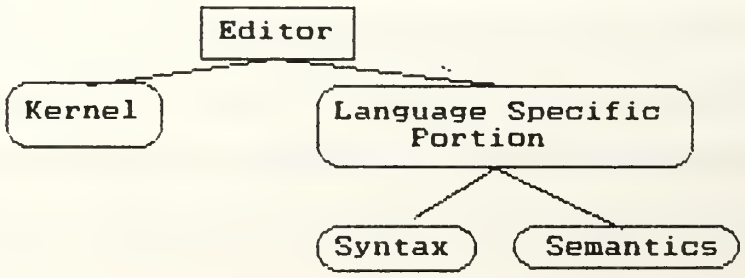

Figure 1 Syntax Directed Editor Generator

directed editors in use today are the Cornell Synthesizer Generator and GANDALF.

1. Cornell Synthesizer Generator

The Cornell Synthesizer Generator creates interactive tools to increase the programming power of programmers. The Synthesizer Generator generates language-based editors using the grammar of the language, its display format, and transformation rules for restructuring programs. The specification of the language's grammar is further subdivided into the language's abstract syntax, context-sensitive relationships, and concrete input syntax. Context-free or context-sensitive errors are determined through knowledge of a language's syntax to establish whether syntax errors have been made locally or globally in order to notify the programmer as soon as they are introduced. Language semantics generate and update code during editing. Incremental generation of code is used to maximize the effectiveness of excess CPU time by evaluating derivation tree segments during editing and to eliminate the need for batch compilation in order to test and debug a program. The Cornell Synthesizer Generator is illustrated in Figure 2. [Ref. 2]

The syntax and semantics of a language provide the relationships within the language which can be used to assist the programmer in the correct usage of the language and prohibit illegal constructs. It is also possible to

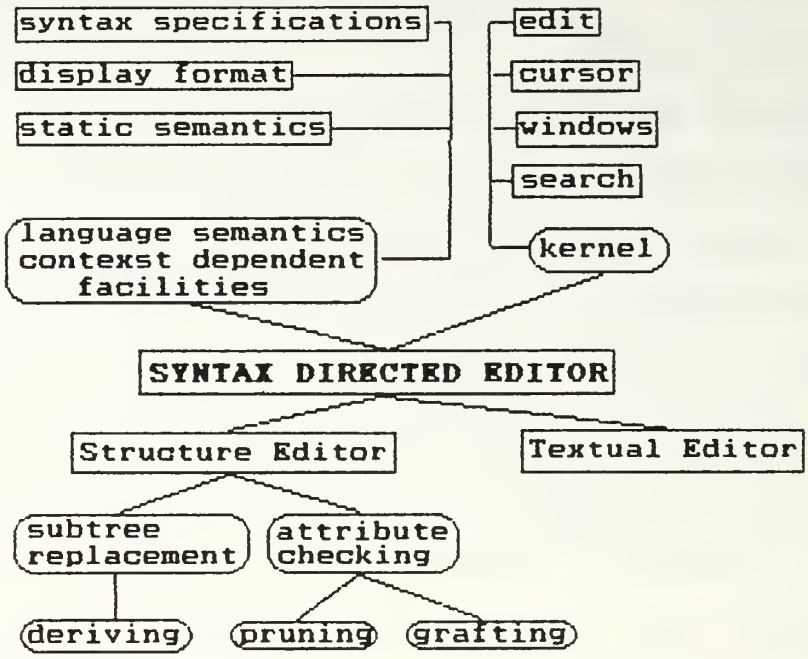

Figure 2 Cornell Synthesizer Generator

relate two languages, say a programming language (Ada) and its specification (PSDL) into a single tool, to implement a particular methodology. [Ref. 4]

An editor created using the Synthesizer Generator creates a program represented by a derivation tree, derived with respect to the context-free grammar (see Figure 3). The derivation tree branches under the direction of a structure editor which supports the top-down construction of hierarchical programs by the insertion of templates into placeholders. The placeholders themselves allow the insertion of additional templates to progressively increase the detail of the program being constructed. The structure editor

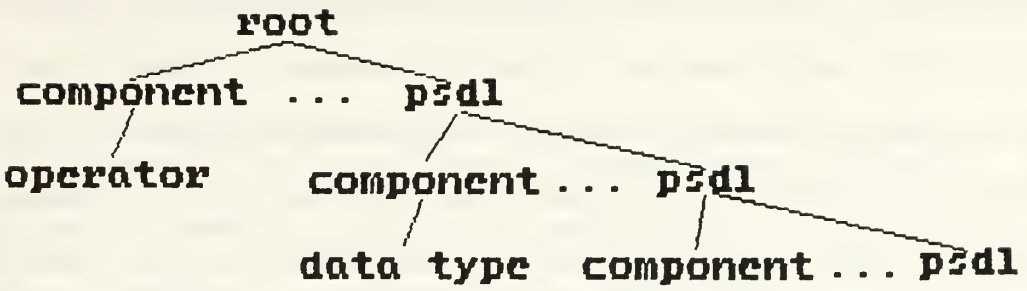

Figure 3 Derivation Tree

supports operations called transformations which are menu-selected legal insertions into a placeholder. Transformations are operations which control changes in the sense that they substitute one legal construct with another legal, although usually more detailed, construct. [Ref. 4]

Templates support programming efficiency by inserting only correctly formed, properly indented program segments which are in the context of the program. Templates match abstract computational units and can be inserted and removed as program units thus promoting abstraction.

The Synthesizer Generator allows creation of hybrid editors in the sense that both character and line oriented operations can be incorporated with structural operations such as template insertion. To allow the use of textual insertions, syntactic categories are created which allow the textual entry, accompanied by the concrete syntax necessary

to recognize the meaning of the input text. Text editing can be limited to restricted constructs, the concrete input syntax for textual input and template-insertion commands for structural input can be context-dependent. [Ref. 4]

The program editors can translate the input into executable code which is also maintained as it is modified. Incremental translation, in addition to continuous compilation, also lends itself to debugging since the executable code can be moved incrementally allowing evaluation only of small portions of reedited code. [Ref. 4]

Immediate computation is caused by incremental algorithms. The Synthesizer Generator bases its algorithm for incremental attribute updating on attribute grammars [Refs. 5,6] and selective recomputation. Attribute grammars are context-free grammars extended by attaching attributes to the nonterminal symbols of a grammar and by supplying attribute equations to define attribute values. Selective recomputation causes only the values that are dependent on changed data to be recomputed. The attribute grammar formalism in the Synthesizer Generator uses declarative specifications, a set of simultaneous equations whose solution order is unspecified and whose data and variables are closely interrelated, to define the immediate error-analysis in its language based editors. Every object created in an editor has a corresponding set of simultaneous

equations from which the editor can calculate the object's error status. [Ref. 4]

In The Synthesizer Generator [Ref. 4], Reps and Teitelbaum state:

> Attribute grammars have several desirable qualities as a notation for specifying language-based editors. A language is specified in a modular fashion by an attribute grammar: syntax is defined by a context-free grammar; attribution is defined in an equally modular fashion, because the arguments to each attribute equation are local to one production. Propagation of attribute values through the derivation tree is not specified explicitly in an attribute grammar; rather, it is implicitly defined by the equations of the grammar and the form of the tree.
>
> The benefit of using attribute grammars to handle the problem of incremental change in language-based editors is that the repropagation of consistent attribute values after a modification to an object is implicit in the formalism. Thus, there is no need for the notions of "undoing a semantic action" or "reversing the side-effects of a previous analysis," which would otherwise be necessary. When an object is modified, consistent relationships among the attributes can be reestablished automatically by incrementally re-solving the system of attribute equations. Consequently, when an editor is specified with an attribute grammar, the method for achieving a consistent state after an editing modification is not part of the specification.
>
> Apart from its use to specify name analysis and type checking, the attribute-grammar formalism provides a basis for specifying a large variety of other computations on tree-structured data, including type inference (as distinct from type checking), code generation, proof checking, and text formatting (including filling and justification, as well as equation formatting). [Ref. 4]

The editor specification addresses both context-free and context-sensitive conditions. As objects are created and modified, the editor continually checks for compliance with the specified context conditions. Context conditions are expressed through the attribution of the object with

attribute equations that express certain constraints. When an object is correctly or incorrectly represented, attributes used in the unparsing specification cause the display to be annotated with values of attribute instances indicating satisfaction or violation of context-dependent constraints. The editor responds to every editing change which affects the user's feedback concerning the validity of his program. This will also inform the user when he has corrected an existing error or when he changes previously correct code to introduce an error, which violates the constraints known to the editor. [Ref. 4] The primitive phyla defined in SSL are defined in Table I.

TABLE I

PRIMITIVE PHYLA

BOOL	Truth values
INT	Integers
REAL	Floating Point
CHAR	Characters
STR	Character strings
PTR	References to SSL values
ATTR	References to attributes
TABLE	Hash tables

Source: [Ref. 7]

Editors are specified by the Synthesizer Specification Language (SSL), a language based on an

attribute grammar and a type definition facility. In SSL, an attribute's type can be an editor supplied built-in type (primitive) or a user defined type constructed as a composite of primitive types. Attribute types and abstract syntax are both defined using the same sort of SSL rules. The result of this double definition scheme is the creation of an abstract syntax tree and attributes on that tree that are fully compatible, that is, all the attributes are elements of a single domain of values. This allows the writing of attribution schemes which themselves create new syntactic objects. [Ref. 4]

2. GANDALF

The GANDALF project not only provides a syntax directed editor generator but it also supports a software development environment. Software development environments provide automated software support that allow simplification of the software development process. Employment of several programmers on a long-term joint project composed of many modules may create problems significantly different from common programming problems. Mechanisms requiring cooperation between programmers are too personality dependent to be adequate. In contrast, system development tools put this support into software thereby enforcing cooperation between programmers. [Ref. 8]

Project management, version control and incremental programming are aspects of software development that are

integrated into a single environment in GANDALF. Through semi-automatic generation, the GANDALF project is involved with creating software development environments that combine both programming and system development environments. [Ref. 8]

a. Project Management SDC (System Development Control System)

Project management is derived from the idea that the information representing the state of a developing software project should be viewed as a set of abstract data types upon which only certain operations may be applied. By maintaining careful type and operation control, the software under development is always in a well-defined state. The SDC (System Development Control System) is a set of programs that provides basic management and communication support within the project team. SDC supports the project team with three levels of program access: readers, project programmers, and project leaders. [Ref. 8]

Generally, SDC commands pertain to multiple source files in a project. The "reserve" and "deposit" commands are the most commonly executed. The user "reserves" a set of source files before making a modification. The "reserve" command locks the selected file and prompts the user to revise them. Actual file Modification is performed using UNIX commands. Following revision, the user returns the source files to the project. This "backs-up" the previous version of the files, unlocks the new version of the

files, and prompts the user for comments reflecting the modifications. The "release" command unlocks the files without incorporating the modifications to the project if the modifications do not satisfy the requirements. Eventually every "reserve" command must be either "deposited " or "released" (this permits rejection of code that does not check out). [Ref. 8]

The system automatically maintains a log of all modifications and prompts authors to explicitly document anything noteworthy. The philosophy is that programmers will document their programs if the process is made sufficiently easy. [Ref. 8]

b. System Version Control (SVCE)

Families of systems sharing subsystems with different versions are interconnected (for example, the source code, the object code, and the documentation) and multiple versions of modules exist in a system. The SVCE checks source files in and out of the system and manages the three kinds of versions:

- o parallel versions, which develop independently from a common specification;
- o successive versions, which develop by building or evolving an existing module as bugs are corrected and features are added;
- o derived versions, which are different instances of a system based on the same source code (optimized code is one example of a derived version). [Ref. 8]

c. Incremental Programming LOIPE (Language Oriented Incremental Programming Environment)

The goal of this project is to support incremental compilation, linking, and loading for the generated languages. Incremental compilation is performed at the grain size of procedures rather than on larger constructs. The extent to which recompilation can be limited depends on the type of modification to the procedure. Recompilation can be limited to the modified procedure when only minor changes are made to local variables and to local control flow. If a procedure's specifications change, then that procedure and the procedures whose scoping depends on it must also be recompiled. [Ref. 8]

Indirect procedure calls are used to call incremental linking and loading routines. The incremental compiler for LOIPE places an index into an entry vector in the object code when a procedure call statement is encountered in a procedure. The indexed entry value then provides the actual location of the called procedure. When a procedure is subsequently revised, the linker and loader update the entry vector to hold the location of the new code. Subsequent procedure calls will execute the new procedure instead of its predecessor. [Ref. 8]

d. Syntax Directed Editing ALOE (A Language Oriented Editor)

The basis for syntax directed editing comes from the conception of a program as structure which is transformed

into text, which is then parsed back into structure. The syntax directed editor relieves the user of the first transformation of structure into text. This eliminates the need for parsing, and in many cases, allows the users to develop ALOE (A Language Oriented Editor) trees directly. [Ref. 8]

These editors provide a good mechanism for replacing the traditional (edit, compile, link, debug) cycle with a more natural and friendly (edit, execute) cycle. ALOE editors are specific instances of a syntax directed editor. [Ref. 8]

All GANDALF ALOE editors share a tree-oriented full-screen user interface. There is a set of language independent commands, including tree traversal and cursor motion commands available in all editors. Language specific details include operators based on the language's grammar which represent language specific structures. For example, Pascal while statements are separately defined for each language specific editor, as are the language specific static semantic checks (type-checking for example). A set of extended commands, may optionally be implemented to support explicit invocation of actions, such as "execute program". [Ref. 8]

The syntax, static semantics, and the set of extended commands must be defined to construct an editor for a particular language. The language, for implementation

purposes, consists of two major parts, syntax and semantics. The syntactic definition of an ALOE editor is divided into two parts: abstract syntax and concrete syntax. (See Figure 4) [Ref. 8]

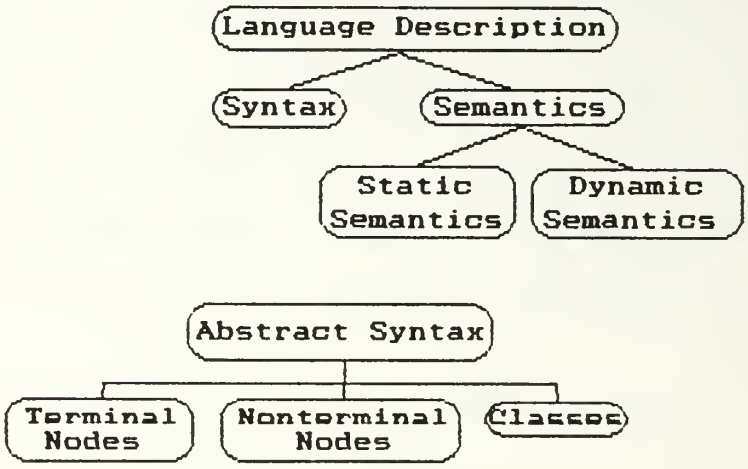

Figure 4 A Language Oriented Editor (ALOE)

The abstract syntax description defines the underlying structures (trees in ALOE) of the language. The abstract syntax is used to define a set of operators and a set of classes. The operators represent node types that can appear in the tree and the classes indicate the ways in which these operators can be composed. Operators may be considered as commands and classes as menus of operators, or commands that may be legally selected. This two level description mechanism permits the generation of a user interface that limits modifications to those which guarantee the syntactic consistency of language trees. The abstract

syntax grammar defines a set of abstract syntax trees that represent all the legal syntactic programs derivable in the specific language. [Ref. 8]

The concrete syntax details the transformation of the abstract syntax to a textual representation suitable for display to the user. The concrete syntax also defines the syntactic sugar (generally redundant keywords) that has traditionally been to support parsing of text to structure in traditional compilers. [Ref. 8]

The semantic support, which includes checking of static semantics and also the definition of extended commands is available as follows:

- o action routines, which are node (operator) specific procedures that are called as a user makes changes to a tree. The action routine for the operator is called with the actual node and the type of the action (for example, create or delete) as parameters whenever a node of that type is created, deleted, or modified. The calling of action routines is automatic and invisible during program manipulation.

- o extended commands which are explicit semantic actions defined by procedures associated with the command names. Extended commands include actions over which the user desires complete control such as execution of a program or reservation of a module. [Ref. 8]

e. The GANDALF Prototype

The goal of the GANDALF prototype was to construct a software development environment based on ALOE generated editors. The GANDALF prototype merges the ideas of SDC, SVCE, and LOIPE into a single integrated environment that produces a syntax directed interface to its users. The functions provided include:

- o project management support, including reserve, deposit, and log creation;
- o system version control support, including parallel and successive versions, and automatic system generation;
- o programming support, including incremental compilation and source-level debugging. [Ref. 8]

f. Designing the Environment

To design an environment, the environment designer supplies the language syntax and semantics to the environment generator. The environment generator then combines the syntax and semantics with the GANDALF system kernel to produce a new environment. [Ref. 9]

The kernel provides such facilities as the command interpreter, editing commands, the user interface, the window manager, and the display package. It is common to all GANDALF editors. This relieves the environment designer of redundant implementation details and provides a uniform user interface in all environments. The kernel is applicable to a broad range of environments. [Ref. 9]

g. An Example of an ALOE

A GANDALF system ALOE (a language oriented editor) is an editor designed for a specific language and is programmed with information about the syntax and semantics of the language. Naturally suited to the domain of programming environments, a programming language editor can provide syntactic and semantic assistance to the user. [Ref. 9]

h. Generating an ALOE

A description of the target environment language syntax, static semantics, and dynamic semantics must be input to the ALOE generator which links the description with the kernel to produce a new ALOE. The kernel supplies the language-independent portion of the environment which is common to all ALOE editors to be melded with the language-dependent portion. [Ref. 9]

Syntax is input in abstract and concrete syntax specifications. The abstract syntax defines the program structure while the concrete syntax defines the physical display of the program. The abstract syntax description is defined in three sections: terminal nodes, nonterminal nodes, and classes. An example illustrating these sections defined in The Evolution of the GANDALF System is shown in Figure 5. [Ref. 9]

The rules defining terminal nodes and nonterminal nodes (productions) in the Pascal-like language are IDENT, TYPEIDENT, INT and REAL. IDENT and TYPEIDENT are defined to follow the lexical rules for a variable {variable}. Static terminals INT and REAL depict a specific string value. Here the curly braces "{" denote the lexical class and the angle braces denote a list. The INT terminal refers to the Pascal string "integer". The VAR_DECL nonterminal decomposes into two children: an identifier "identlist" and a type "type".

o Terminals:

```
IDENT = {variable}
TYPEIDENT = {variable}
INT = {static}
REAL = {static}
```

o Nonterminals:

```
VAR_DECLS = <decl>
VAR_DECL = identlist type;
IDENTLIST = <ident>
```

o Classes:

```
decl = VAR_DECL
ident = IDENT
identlist = IDENTLIST
type = TYPEIDENT REAL INT
```

Figure 5 Grammar for Pascal-like Variable Declarations

The children are specified by a class, "identlist" to IDENTLIST and "type" to either TYPEIDENT, REAL, or INT. The class represents a menu of possible nodes (sometimes a menu of one) that could be inserted as the child of that nonterminal, that is, for "type", there are three alternatives. From the editor designer's perspective, a class can be considered a type union with the nonterminals describing the relationships between the various type unions. The "type" class specifies that a "type" may be represented by one of the following kinds of nodes: TYPEIDENT, REAL, or INT. The VAR_DECLS and IDENTLIST nonterminals represent lists of elements where each element of the list comes from

the same class. Terminals, nonterminals, and classes are defined in Figure 6. [Ref. 9]

terminal -	primitive symbol in a language
nonterminal -	a symbol, which when applied to an input symbol, points to a new production (children are classes)
class -	menu of possible nodes (terminals or nonterminals)

Figure 6 Abstract Syntax Description

The next portion of the language specification is the static semantics. Static semantics refer to the type of semantics that can be checked by examination of the program. It encompasses type checking, checking of scope rules (if the language is statically scoped) and the like. Action routines are written on selected nonterminal and terminal productions in the grammar. [Ref. 9]

The final section of the language specification is the dynamic semantics. The dynamic semantics provide the tools necessary to execute a program. An incremental compiler or an interpreter could be built-in to permit program execution from any point in the program. Extended commands are used to add dynamic semantics to an ALOE. An extended command is a command written by the editor designer which is added to the environment and appears as a kernel command to the user. [Ref. 9]

i. Programming Environments

GANDALF programming environments provide a highly interactive knowledge-based environment for the programmer. The intent of the system is support of small projects. The knowledge programmed into the environments includes a minimum of the syntax and static semantics of the programming language. More advanced implementations of GANDALF environments include dynamic semantics. A well designed programming environment confirms the program's syntactic and static semantic correctness and provides the ability to run a partially constructed program at any time during the creation of a program. [Ref. 9]

j. User Interface

The user interface is the component of any environment which most significantly affects user acceptance. The user interface is based on the kernel and therefore provides a uniform presentation for all ALOE editors regardless of the language. The kernel allows tailoring of primitives according to the language specification, yet maintains its style of user interaction. [Ref. 9]

3. Evaluation and Comparison of the Cornell Synthesizer and GANDALF

The GANDALF tool provides an environment which allows team development of a software project. It features both programming and system development environments which exceed the scope of the Cornell Synthesizer Generator which does not

support system development environments. Since the editor in CAPS is only a part of a system that has its own built-in system development features, this feature is not relevant in this application.

GANDALF and the Cornell Synthesizer Generator are both based on derivation trees and differ primarily in the way they deal with attributes. GANDALF has adopted the Cornell Synthesizer's use of multiple trees to improve modularity. The Cornell Synthesizer Generator uses simultaneous equations to continually update the attribute while GANDALF depends largely on static checking. Both GANDALF and the Cornell Synthesizer Generator have the ability to adjust the ratio of structural editing to textual entry.

Due primarily to time constraints, the Cornell Synthesizer Generator was chosen before any extensive analysis of the GANDALF ALOE generator was performed. The Pascal Editor included in the Cornell Synthesizer Generator was tested and found to be an excellent editor.

Semantic checking is performed more naturally using the Cornell Synthesizer Generator's attribute equations than the writing of action routines to perform dynamic run-time semantics. The Cornell Synthesizer Generator also has capabilities for enhancing CAPS beyond the generation of a capable syntax directed editor. The Synthesizer Generator

also has the power to incorporate simple static timing constraints and limited translation of PSDL directly into Ada. Control Abstractions may also be checked using the Synthesizer Generator.

II. A DESCRIPTION OF THE PROTOTYPE SYSTEM DESIGN LANGUAGE (PSDL)

A. PROTOTYPE SYSTEM DESIGN LANGUAGE (PSDL)

PSDL was designed specifically for the Computer Aided Prototyping System (CAPS). Many features were considered in the design of PSDL to meet both the requirements of a design language and a prototyping language. Real-time constraints, executability, and control aspects were incorporated to allow the modeling of the actual performance of the system being designed. To facilitate the use of the language, the following features and requirements were designed into it:

- o based on a simple computational model
- o executable
- o easy to use
- o support of hierarchically structured prototypes
- o support both specification and design
- o data base accessible/addressable modules
- o support for formal and informal module specification
- o support of data
- o function and control abstractions
- o support of real-time systems.

PSDL specifies the characteristics and functions of components in the prototype and reusable components in a software data base. It is also used to establish the connections between the various operators in a prototype. [Refs. 3, 10]

1. Model

PSDL is based on a graphical model consisting of operators that communicate between themselves by means of data streams which convey information using a fixed abstract data type. Each operator corresponds to an Ada subprogram in the final product. The computational model is an augmented graph

$$G = (V,E, T(v), C(v))$$

where V is the set of vertices (operators), E is the set of edges (data streams), T(v) is the timing constraint associated with the operator at that vertex, and C(v) represents the control constraints associated with the vertex (see Figure 7). [Ref. 3]

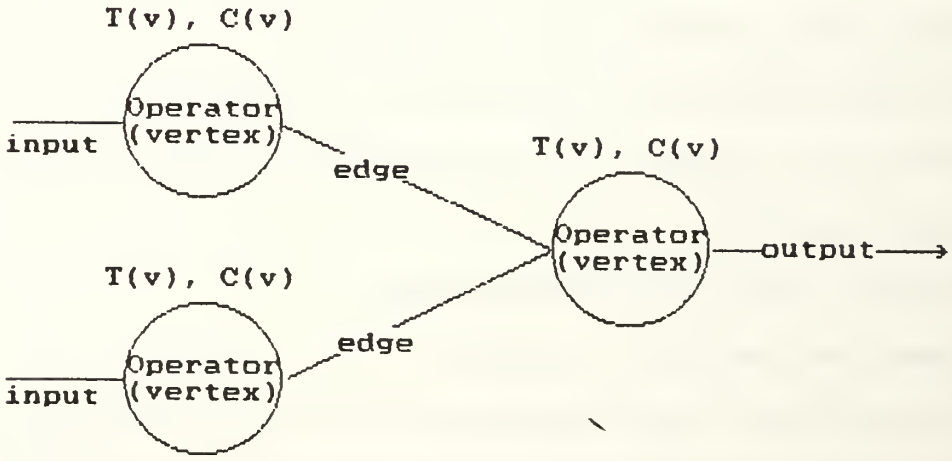

Figure 7 PSDL Computational Model

2. Operators

The operators fall into two categories, functions and state machines. Operators have potentially several inputs and outputs which are capable of delivering or transmitting up to one data object at each operator firing. The output values are determined by the inputs and the state of the operator at the time of firing. The concept of an operator supports top-down design in that operators support different levels of abstraction, for example, one operator may represent a collection of smaller operators or the same operator may be decomposed into simpler (less abstract) or simplest (atomic) representations. [Refs. 3, 10]

3. Communicators

The data stream is the link by which two operators communicate and pass data. Data values are transmitted sequentially within data streams. The two types of data streams, data flow streams and sampled streams, present data to their respective operators. The data flow stream guarantees delivery of the data sequentially in a FIFO queue while the sampled stream presents only the most current data value in its memory. The data flow stream is used when all information is essential and the sampled stream is used when it is likely that information may come in faster than a real-time ability to process it. Stream types are not explicitly declared but rather, are determined by the type of the destination operator. [Refs. 3, 10]

Exceptions are built-in data types that recognize whether data values are normal (do not belong to the EXCEPTIONS data type) or are members of named EXCEPTIONS. Exceptions are exceptional situations requiring a special response and are values belonging to the EXCEPTION abstract data type. Exception values are treated as normal values in terms of their ability to be sent to other operators in data streams. Exceptions can be produced in PSDL and Ada. Exceptions produced in an Ada component which are not handled internally are converted to PSDL exceptions and transmitted to all the component's output streams, subject to any output guards associated with that stream. [Refs. 3, 10]

4. Abstractions

Abstractions are fundamental in order to express complex systems succinctly and simply. Abstraction is specifically supported in the following areas: operators, data, and controls. [Ref. 3]

The operator is an abstraction of either a function or a state machine. The abstract operator is used to model software systems as either an entity whose outputs are determined solely by its inputs (a functional abstraction), or determined by the combination of its inputs and its internal state (a state machine abstraction). State Machine abstractions are denoted by the keyword STATE. The operator abstraction can be implemented by decomposition into smaller composite operators or atomic operators. Operators are

composed of two primary parts, the SPECIFICATION and IMPLEMENTATION. The SPECIFICATION includes the interface parameters, timing characteristics, and the formal and informal behavioral descriptions of the operator. These attributes can be used to isolate a specific operator and can be used as keys in a software database. The IMPLEMENTATION specifies whether an operator is implemented in Ada (or other underlying programming language for other systems), or whether the operator is made up of composite parts. [Refs. 3, 10]

Data abstractions contribute to the minimization of coupling by separating the behavior of a data type from its representation. The consequences of this data abstraction are the ability to operate the prototype with incomplete sources of data and also to allow totally different data representations in the prototype and the actual system. Data abstraction also allows the data interfaces to be described independently of the representation of the data, so that the interfaces for the operations on the data can be the same in the prototype as in the intended system. This commonality of the data interface is important in the validation of the final system because it gives a basis to verify the structures against the prototype. [Refs. 3, 10]

Data types in PSDL are immutable (do not have internal states). This prevents communication by way of side effects. They include the built-in data types defined in

Ada, abstract types defined by the user, and types constructed using the PSDL data types. Like the operator, the PSDL data type is defined using the components SPECIFICATION and IMPLEMENTATION. [Refs. 3, 10]

Control abstractions are used to determine whether operators can be run in parallel or must be run in sequence and allow for the scheduling of the order of execution. Naturally, within a composite operator there are also control abstractions. Another feature of the control abstraction is the provision for conditional execution using triggering conditions and conditional outputs. The control abstraction is regulated by control constraints, a non-algorithmic mechanism that is specified to be PERIODIC or SPORADIC, has a triggering condition, and has output guards. The periodic and sporadic operators refer to the scheduling of an operator, either within a specified period or when triggered respectively. Triggers are data values specified to cause an operator to fire. Every operator must have a data trigger or a period and may have both. The combination of control constraints also defines the stream types for the data streams in the enhanced data flow diagram. The stream type of a data flow stream is determined by the data trigger of the operator to which it is an input, not the operator originating the data flow stream. The data trigger ALL specifies a data flow stream, and the data trigger SOME specifies a sampled stream. [Refs. 3, 10]

5. Timing

Timing constraints express the essential timing requirements necessary to operate within real-time constraints. Modules can be specified with MAXIMUM EXECUTION TIME, MAXIMUM RESPONSE TIME, and MINIMUM CALLING PERIOD. The MAXIMUM EXECUTION TIME corresponds to the amount of time that a module has to execute and may apply to any PSDL operator. The MAXIMUM RESPONSE TIME corresponds to the allowable time a sporadic operator has to output data from the time it was triggered by the input data. The MINIMUM CALLING PERIOD constrains the time interval between input data streams. [Refs. 3, 10]

6. Hierarchy

Hierarchical structure is realized in the concept of the composite operator. The constraints necessary to decompose operators require clear communication via data streams at the different hierarchical levels. The input streams and the output streams into and out of a component operator must naturally be accounted for in the parent composite operator and every data stream going to or from a composite operator must be a valid input or output associated with at least one of the decomposed component operators. Timing must also be constrained so that the execution and response times of the component operators do not exceed the MAXIMUM EXECUTION TIME or MAXIMUM RESPONSE TIME of the parent composite operators. PSDL is designed so that the component

operators downwardly inherit the period, minimum calling period, and the stream types of the streams crossing the boundary of the composite. The composite operators upwardly inherit exceptions and integrate the output data streams. [Refs. 3, 10]

B. ATTRIBUTE GRAMMARS

Attribute grammars allow both the syntax and semantics of a language to be specified. Attribute grammars are context free grammars which are extended by attaching attributes to the grammar symbols. In the specification of a language, attribute grammars allow top-down rules to be interspersed with bottom-up rules. [Ref. 11]

An attribute grammar consists of a set of productions that contains sets of semantic equations. Each semantic equation defines one attribute to be the value of a semantic function applied to other attributes in the production. The semantic functions define values for the synthesized attributes of the left-hand side nonterminals or the inherited attributes of the right-hand side symbols. The attributes of a symbol are divided into two disjoint sets:

- o synthesized attributes which pass information up the derivation tree
- o inherited attributes which pass information down the derivation tree. [Ref. 11]

A language symbol is represented by a derivation tree node which defines a set of attribute instances. (See Figure 8.) Attribute instances correspond to the attributes of the

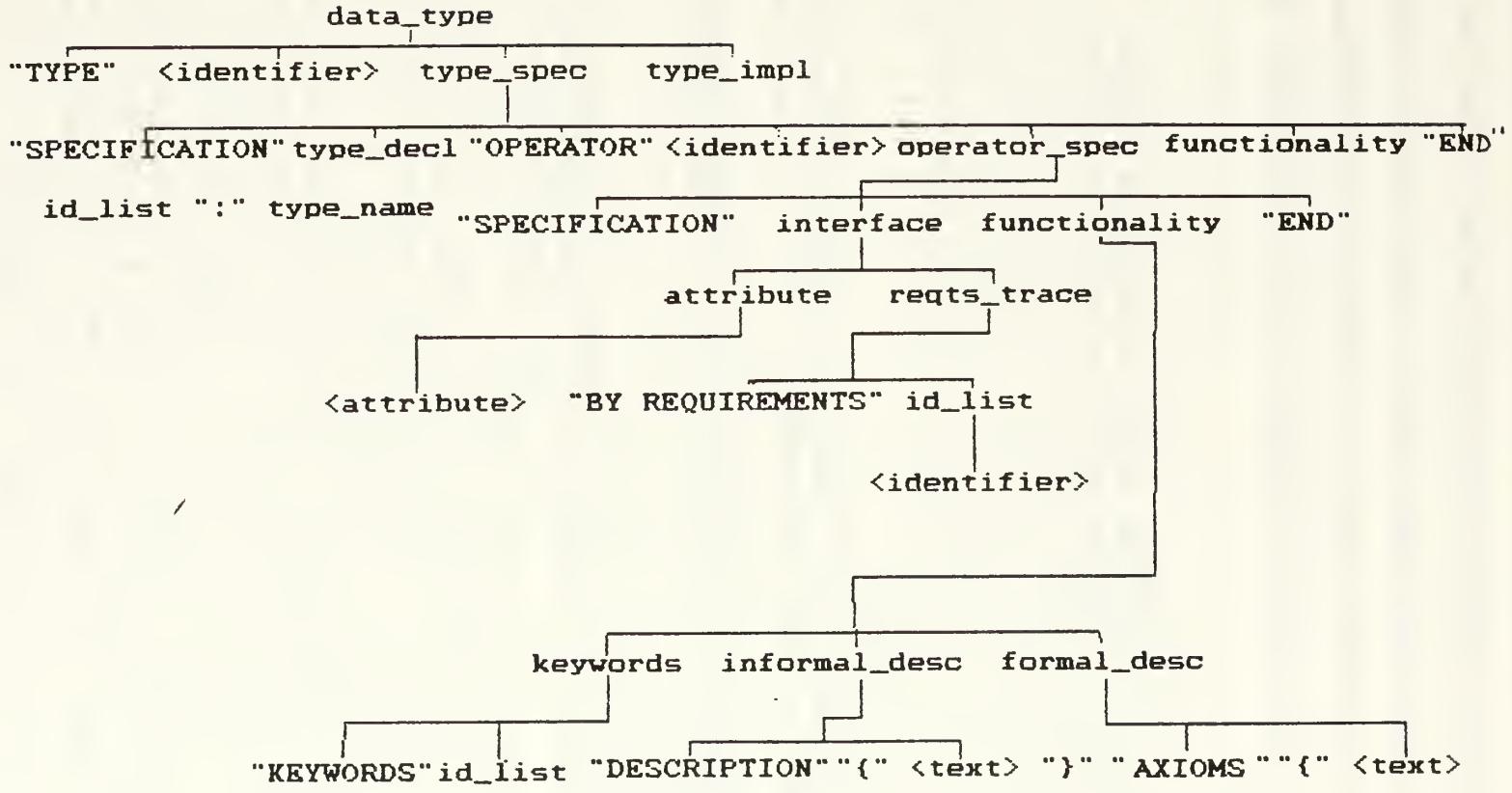

Figure 8 Derivation Tree

syntactic category of the parent derivation tree node (symbol). Since attribute values flow in both directions, it is necessary to impose conditions to ensure that no attribute instances are circularly defined. Syntax and semantics are defined in modules using a context free grammar and the semantics are defined based on the syntactic structure. [Ref. 11]

C. OVERVIEW OF COMPUTER AIDED PROTOTYPING SYSTEM

The Computer Aided Prototyping System (CAPS) is a system designed to cope with ever larger software systems. Conceptually, it follows the needs presented by current software shortcomings.

- o It addresses the concept of communications between the customer and the programming team
- o It addresses the problem of the project scope exceeding the capacity of a single person to comprehend the entire project
- o It incorporates reusability of software, it promotes modularity, it recognizes real-time constraints
- o It incorporates human factors into its design. [Ref. 12]

Essentially, the CAPS system allows the customer and the project manager to express the needs of the system by iteratively constructing refined prototypes that demonstrate the appearance and simulate the performance of the end product. After the design is approved, the prototype, written in PSDL, is used to index a data base of reusable software [Ref. 13] and applicable program segments are substituted for equivalent PSDL operators. The remaining

PSDL operators can be broken down into composite operators using either the PSDL syntax directed editor or the graphics editor. [Ref. 14] Decomposition of PSDL operators continues until all the PSDL operators have been matched with equivalent Ada code segments or until the operators are atomic, at which point the operator is translated into Ada manually. [Refs. 3, 12, 15]

Motivation for CAPS stems from a trend toward increased software demand for larger and more complex systems. As the scope of projects increases, there is an accompanying need for improved productivity and reliability of software. Successful communication between the customer and programming organizations is necessary and is dependent on requirements analyses and validation processes. This process communicates the customer's needs and minimizes the number of revisions necessary to develop the desired product. It also validates feasibility studies, confirms viability, and enhances cost estimation. [Ref. 1]

The CAPS method uses rapid prototyping to create an executable model of the intended system. The model is built by establishing a modularized skeleton, which validates the timing constraints [Ref. 16] and systematically defines the module interfaces (input, output formats), thereby maintaining traceability to the project's requirements. This model uses either a problem oriented top-down design to focus on the critical sections of the problem or it may model the

entire system. Once an executable prototype is built, requirement refinement is verified with the customer and the prototyping/refinement process is reiterated until the model performs acceptably. [Refs. 3, 12]

The capabilities in the CAPS system include syntax directed and graphical editing, reuse of software using a PSDL keyed software base, code generation to interconnect modules, static and dynamic task scheduling, debugging, design control using a design data base and limited automatic translation from PSDL into Ada. [Refs. 3, 12-18]

The rapid prototyping quickly produces a program skeleton that is easy to understand, analyze and revise and in which timing constraints are demonstrated in terms of whether components will be able to perform their design functions in the time allotted. [Refs. 3, 12]

D. BACKUS-NAUR FORM FOR PSDL

The Backus-Naur form for PSDL grammar is contained in Appendix A.

III. THEORY OF CORNELL SYNTHESIZER GENERATOR

A. TRANSLATOR AND KERNEL

The Synthesizer Generator is composed of two parts. The first part is a translator that takes a Synthesizer Specification Language (SSL) coding as input and produces various tables as output. The second part is an editor kernel that consists of an attributed-tree data-type and a driver for interactively manipulating attributed trees. The kernel takes input from the keyboard and mouse and executes appropriate operations on the current tree. [Ref. 7]

B. BUFFERS

The editor generated places objects into a collection of named buffers. Normally, each edited file has a unique buffer. The objects contained in buffers are called terms and are derivation trees with respect to the abstract syntax of the language. The nodes of a term are instances of operators and the subtrees of a node are the operator's arguments, themselves terms. Each term has a textual output representation. [Refs. 4, 7]

Each buffer has a selection, the area of interest to be edited. The selection can be chosen by tree walking commands such as forward preorder, backward preorder, forward sibling, backward sibling, and ascend to parent. [Refs. 4, 7]

C. TRANSFORMATIONS

Each editing transaction replaces a subterm or sublist. A transformation determines a replacement value for the selected subterm as a function of its current value. Transformations are enabled or disabled in accordance with the matching of its pattern with that of the selection. Enabled transformations are substitutions that can be made for certain values, for example, <component> could be replaced by the following two alternatives:

"TYPE <identifier>
SPECIFICATION <identifier> : <type name>
OPERATOR <identifier>
SPECIFICATION
<attribute><requirements trace>
<functionality>
END
<functionality>
END
IMPLEMENTATION <type implementation>"

or

"OPERATOR" <identifier>
SPECIFICATION
<interface>
<functionality>
END
IMPLEMENTATION
<operator implementation>".

Transformations cannot introduce context-free syntax errors since their definitions are also type-checked when the Synthesizer Specification Language (SSL) specification is compiled into an editor. During compilation, the grammar is checked for consistency. [Ref. 11]

D. EDITING

The Synthesizer Generator editors have a language dependent part and a set of system commands which are language independent and strictly control editor functions, for example, cutting and pasting. To ensure connectivity of the tree structure, the editor inserts placeholders to replace any deleted subterms. [Ref. 11]

The textual display of the selection can be edited like a typical screen-oriented text editor. The text, initially is placed into a text buffer upon the first action that implies textual modification. The text buffer is then displayed on the screen instead of the selection, causing its existence to be practically invisible to the user. Within the text buffer, editing is unrestricted. A character selection, identifying the location that text changes occur, can be positioned by moving the cursor or a mouse. When the term selection is moved away from its current location, the text is parsed with respect to the concrete input syntax of the current selection context. If a syntax error is detected it must be corrected before the contents of the text buffer can be translated into a term, which in turn, replaces the original subterm and is displayed according to its output representation. The generated effect may be to format the text on the screen or transform it into an equivalent representation. [Ref. 11]

E. ATTRIBUTION

A term in a buffer is attributed, which means it has associated with it computed values that characterize the term. Every time a buffer is modified, the attributes are revised so that they remain current and form a consistent database of derived information. Selected attribute values are displayed to the user as part of the output presentation during the editing session. [Ref. 11]

F. FILE REPRESENTATION

The buffer contents can be written into one of two types of files: an abstract structure file which records the abstract structure of the terms or a textual file which records the textual output representation. Generation of a text file is possible only if the concrete input syntax is complete. [Ref. 11]

G. PRACTICAL MATTERS CONCERNING IMPLEMENTATION

The novice editor designer is advised to consider the costs in time and effort, and if determined to be justified, to proceed in a modular fashion. The modules should reflect the breakdown in SSL: abstract syntax, attribute declarations, unparsing declarations, concrete input syntax, and transformation declarations. Ultimately, to reduce end-user frustration, the editor should be capable of receiving as input both text entry and template insertion. [Ref. 4]

The order of editor module implementation should be seriously considered in that some portions of editor development are more intuitive than others. [Ref. 4]

Initially, it is essential to extract a manageable subset of the language which represents the essence of the language. In substance, this means drawing a line after a portion of the grammar and either ignoring or selecting very limited portions of the remainder of the language. In PSDL, the first cut reduced the language to a definition of either an operator or a data type as shown in Figure 9.

```
<psdl> ::= <component>*
<component> ::= data_type
        | operator
<data_type> ::= "type" id type_spec type_impl
<operator> ::= "operator" id operator_spec operator_impl
```

Figure 9 Abbreviated PSDL in Backus-Naur Form

The Backus Naur form (BNF) of the grammar should be translated into an abstract syntax. The original grammar may not be defined specifically for translation into a syntax directed editor and may contain extra syntactic categories designed to reduce ambiguities to allow deterministic and orderly parsing so that it may have to be restructured when defining the abstract syntax. The edited object must be hierarchically structured so that the abstract syntax tree is understandable and reasonable from the editor user's point of view. The naturalness of the decomposition of the edited

object will determine the feel of the editor because the user will be cutting and pasting subtrees formed under these rules. [Ref. 4]

Over-specification or definition of unnecessary syntactic distinctions will impede rapid programming by requiring the user to split hairs to get the editor to accept a valid input. Syntactic sugar is not needed in the definition of the abstract syntax. Only terminals which contain semantic meaning should be retained in the abstract syntax trees. The operator in a production is sufficient to identify the specific usage of the left-hand-side-phylum (the set of terms derived from the given nonterminal symbol). Other keywords or separators can usually be dropped from the abstract syntax. [Ref. 4]

Since the abstract syntax tree is the only intermediate storage medium for the editor objects, different textual representations of the same object will be stored identically, and will thus be represented as the same object when retrieved from the abstract syntax tree. The distinction between semantically identical objects can be, but should not be retained. A standard form is the preferred output of a generated editor; for example, the number 17 should be output whether the integer 0017 is entered or the number 17. [Ref. 4]

Attribution may force a change in the abstract syntax depending on the overloading in the grammar. For example, if

an identifer may represent different types, then the semantic analysis becomes context-dependent. [Ref. 4]

Initially, the unparsing declarations should be kept simple, being intricate enough only to further debug the generated editor. Consideration of alternate unparsing schemes, optional line breaks, and context-dependent displays should be postponed until the overall development of the editor is in the later stages. It is advisable to indicate the location of attribute values. Initially it is best to specify every possible resting place (the node at which the apex of a selection can rest) and weed out the undesirable resting places later. [Ref. 4]

Transformations should be designed to permit top-down derivation of an object. Initially, only transformations that correspond directly with productions of the grammar should be defined. These transformations are known as template transformations because when one is invoked, it generates a template into which additional components may be inserted. [Ref. 4]

Templates are associated with productions of the form :

$$X_0 : \text{operator } (X_1 \ldots X_n)$$

and correspond to the operator and the n terms. The form for a template transformation corresponding to the above production is:

transform X_0
    on transformation-name<X_0> : operator(<X_1>, ... ,<X_n>);

The placeholder $<X_i>$ terms may be substituted with the completing $[X_i]$ terms, the distinction being that the template reflects the placeholder when specified using the "<>" brackets. Normally template transformations are not necessary for list phyla and optional phyla due to built-in transformations for them in the kernel portion of the editor. Lexemes (keywords and punctuation) do not use template transformations. Lexeme insertion is accomplished in the concrete syntax. [Ref. 4]

At this point, it is recommended that an editor be generated to verify, debug, and confirm choices made up to this point. [Ref. 4]

Given satisfactory results at this stage, a minimal concrete syntax should be defined to allow text input. Initially, provision should be made for the entry of lexemes and simple expressions. The parser can be elaborated later and must be elaborated since the ability to read objects from existing text files is necessary within the context of the CAPS environment.

IV. CONCEPTUAL DESIGN AND SOFTWARE ENGINEERING OF A PSDL SYNTAX DIRECTED EDITOR

In the design of the syntax directed editor, a software engineering approach was used to decompose the system into manageable components. [Ref. 1]

A. REQUIREMENTS ANALYSIS

The purpose of the PSDL syntax directed editor is to ease the programming burden of prototype development by providing an editor with the following features:

- o built-in grammar
- o templates available for common constructs
- o formats the input properly
- o performs syntactic and semantic analysis.

The constraints on the editor include the use of a Sun window based UNIX environment, use of the Cornell Synthesizer Generator or GANDALF ALOE generator, and a limited amount of time available due to the periodic rotation of students at the Naval Postgraduate School.

1. Operating Environment

The syntax directed editor can run in either the UNIX or Sun environment.

2. System Goals

Provide a Syntax Directed Editor for the front end of a Computer Aided Prototyping System. Provide a user

friendly Prototype System Design Language (PSDL) syntax directed editor which gives the user a means of writing, editing, and modifying syntactically correct Prototype System Design Language (PSDL) code which includes limited semantic checking.

3. Performance Constraints

The editor should never be unresponsive to the user for greater than one-half second.

4. Implementation Constraints

The hardware decision is to use Sun work stations. Future development may include porting to other systems including PC's, but currently, various portions of Computer Aided Prototyping System are written in different languages thereby complicating portability.

The UNIX operating system is the unifying environment common to all the subsystems within CAPS. The C programming language is the basis for the Cornell Synthesizer Generator software.

The PSDL language-based editor is to be implemented using the Cornell Synthesizer Generator. The specification for the language includes context-free abstract syntax, context sensitive relationships, the display format, and concrete input syntax. The Cornell Synthesizer Generator editor can be programmed to enforce the syntax and static semantics of a particular language. The Synthesizer Generator produces editors which form programs as consistently

attributed derivation trees. As the program is edited, one well-formed tree is changed into another well-formed tree. As this transformation occurs, some of the attributes may lose their consistency of values and in order to compensate, incremental analysis is performed by updating attribute values throughout the tree to correct for each modification. If editing modifies an object to the point that context-dependent constraints are violated, an error message will be generated. [Ref. 7]

Editor specifications are written in SSL, Synthesizer Specification Language, which is based on the ideas of term algebra and an attribute grammar. [Ref. 7]

5. Resource Constraints

At the Naval Postgraduate School, availability of Sun terminals which was once inadequate is now improving. The primary resource constraints are time to dedicate to this project and the turn around times incumbent in the Navy Supply System.

6. External Interfaces

The editor is accessed from a UNIX environment by invoking the editor name (psdl.syn) optionally followed by the filename of new or existing PSDL (UNIX) files. The editor has the capability to rename and create new files while a file is loaded in the editor.

B. FUNCTIONAL SPECIFICATIONS

The PSDL syntax directed editor can be classified as an abstract state machine which interfaces with the CAPS system only through the user interface [Ref. 15]. To ensure compatibility with the Graphic Editor [Ref. 14], the syntax directed editor will be implemented on the Sun work station. The software package, either Cornell Program Synthesizer or GANDALF, will determine the actual display.

The editor can receive input as UNIX files or direct operator input; it notifies the user of incorrect syntax and allows only correct Prototype System Design Language code to be written prior to continuing the program. On demand, the editor will prompt the user with legal syntactically correct alternatives based on his location in the program. The editor outputs UNIX files which represent the abstract structure of the program or the textual format.

Other mechanisms in the CAPS system indirectly communicate with the syntax directed editor via the user interface.

The editor will support only one programmer in a PSDL program at a time, although multiple programmers may edit different programs simultaneously.

Program editing time will be considerably slower than normal keyboard entry although actual time spent programming non-trivial programs should be reduced due to reduced error rates.

The characteristics of the system are described in the User Guide to the Prototyping Language Editor (Chapter V).

C. ARCHITECTURAL DESIGN

The decomposition of the syntax directed editor is a function of the software package used to generate the editor. In both GANDALF and the Cornell Synthesizer Generator, there is a major subdivision between the kernel which represents the presentation and non-language specific details of the editor, and the language-dependent portion which takes the grammar of the language and the formatting instructions to customize the editor to a particular language.

The Synthesizer Generator was chosen to implement the PSDL editor; it is composed of two parts: a translator that takes a Synthesizer Specification Language (SSL) specification as input and produces various tables as output and an editor kernel that consists of an attributed-tree data-type and a driver for interactively manipulating attributed trees. The kernel takes input from the keyboard and mouse and executes appropriate operations on the current tree. [Refs. 4, 7, 11]

Within the language-dependent context of the editor generators, further modularization is effected by the further subdivision into aspects of the language. The grammar is fed into the Synthesizer Generator using Synthesizer Specification Language (SSL) specifications. Synthesizer Specification Language (SSL) specifies the root of the

derivation tree, the abstract syntax, the attribution rules, the unparsing rules, the template commands, the allowable transformations, the lexical syntax, and the parsing syntax to describe the various grammars, type dependencies, semantic rules and embellishments (see Figure 10). [Refs. 4, 7, 11]

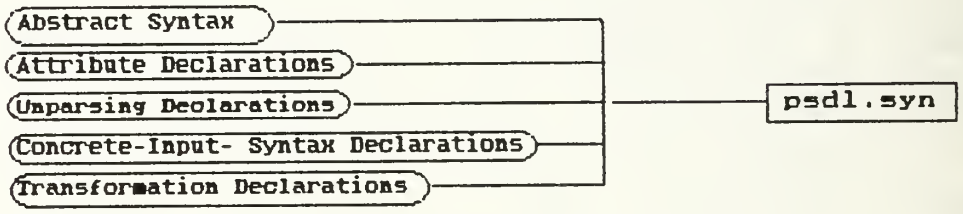

Figure 10 SSL Modules in psdl.syn

D. IMPLEMENTATION

The specification of an editor in SSL is a five part process that includes: abstract syntax declarations, attribute declarations and attribute equations, unparsing declarations, concrete-input-syntax declarations, and transformation declarations. The specification of abstract syntax defines the basic structure of the language. The attribute equations and attribute declarations are used to track the context-sensitive relationships in the language.

The unparsing declarations format the program output in terms of output display. The concrete-input-syntax declarations evaluate the textual input. The transformation declarations provide legal program options for restructuring programs. [Ref. 4]

SSL allows infix expressions, function applications, conditional expressions, the construction of new terms, and conditional pattern matching expressions (with expression). Conditional pattern matching can be introduced using a base-type declaration to define new value representation. The base-type declaration references the name of a C type and six operations on that type. The six base-type operations are: comparison, conversion from ascii, conversion to ascii, incrementation of a reference count, decrementation of a reference count, and generation of a default base-type value. These operations and the type definitions are written in the C language. [Refs. 4, 7]

1. Abstract Syntax Specification

The specification of abstract syntax defines the core of an editor. The abstract syntax is input as a set of grammar rules that are used as the basis of the representation of an edited object as a derivation tree (syntax tree). Grammar rules are defined as productions in the form

$$x_0 : op(x_1\ x_2\ \ldots x_k);$$

where op is the name of an operator and each of the x_i is a phylum, a nonterminal of the grammar. The PSDL construct:

```
psdl_impl:  PsdlImpl(data_flow_diagram
            optional_streams
            optional_timers
            optional_control_constraints
            optional_informal_desc)
            ;
```

represents the psdl_impl production where PsdlImpl is the operator name and optional_streams, optional_timers, optional_control_constraints, and optional_informal_desc are nonterminals (phyla) in the grammar which are themselves subsequently defined. The Backus Naur (BNF) representation for this production is:

```
<psdl_impl> ::= data_flow_diagram
                [ streams ]
                [ timers ]
                [ control_constraints ]
                [ informal_desc ]
                "end"
```

Each phylum represents a set of derivation trees derivable from the nonterminal represented by the phylum. Each of these derivation trees, in turn, are called terms. Terms express the underlying meaning of a phylum but may be displayed differently, transformed into a different display presentation by the unparsing rules. [Ref. 4]

In the abstract syntax representation, the root phylum initiates the derivation tree and all phyla derived from it are defined in terms of it. The root phylum in PSDL, from the BNF <psdl> ::= <component>* is: root psdl. Each subsequent phylum has a completing operator, which defines a

default representation using completing terms in the absence of more complete specification by the editor user. This ensures that a derivation tree is always complete, regardless of the state of composition, that is, the program has legal defaults which stand until replaced by the editor user. [Ref. 4]

The abstract syntax also allows the definition of some phyla as lists. A list is a construct that allows one or more objects to be entered, while an optional list allows zero or more objects to be entered. The BNF representation of the expression_list:

<expression_list> ::= expression <"," expression>*

is translated into the following partial SSL representation.

```
list expression_list;
expression_list :   ExprListNil()
        |   ExprListPair(expression expression_list)
        ;
```

The comma in the BNF notation will be defined in the unparsing declarations in another section or module. An optional list construct in PSDL, for example the root phylum

<psdl> ::= <component>*, translated into SSL is:

```
optional list psdl;
psdl    : PsdlNil() /* This is a completing term */
    |PsdlPair(component psdl). [Ref. 4]
```

The editor treatment of lists includes special built-in routines for manipulating lists including the forward-with-optionals command. Lists must be defined as a nullary operator and a right recursive binary operator. Lists have a default representation based on the completing

term. The completing term of a list phylum is the singleton list constructed by applying the binary operator, in this case PsdlPair, to the completing term of its left argument phylum and to the list's nullary operator, psdl. The completing terms are artificially added by the editor to satisfy the attribute tree until substitution by a legal PSDL phylum declaration. [Ref. 4]

The first operator declared for a phylum, such as the operator DaTyComp of phylum component and the operator DaTy of the phylum data_type, is called the completing operator and is used to construct a default representative for the phylum, called the completing term. The abstract syntax representing the abbreviated grammar is shown in Figure 11. A more complete approximation of the abstract syntax for PSDL is contained in Appendix B.

2. Attributes and Attribute Equations

After the grammar rules are implemented to define the abstract syntax, attribute equations are added to define the static inferences about the objects being edited. The static inference conditions checked are whether a declaration is supplied for identifiers, if an identifier is multiply declared, and if the constituent of expressions are type-compatible. During program editing, these conditions are incrementally checked. Comments, defined as attributes, are used to report violations of these. When an error is present,

```
root psdl;              /* tree root */

optional list psdl;
psdl : PsdlNil()    /* This is a completing term */
     |PsdlPair(component psdl);
     /* allows recursive addition of components
       within the program */

component: EmptyComp()    /* required as a completing
                          term  */
     | DaTyComp(data_type)     /* function calls */
     | OpTyComp(operator);


data_type: DaTy(identifier identifier identifier);
operator: Op(identifier identifier identifier);
        /* notice   that the keyword "operator" will
         have to be inserted in the actual text of
        the  output program */

identifier: IdentifierNull()
        | Identifer(IDENTIFIER);
```

Figure 11 Abstract Syntax for Abbreviated PSDL Grammar

it satisfies the attribute value corresponding to an error message, similarly, when no error is present the corresponding attribute value is the null string. Since the conditions are incrementally checked, correction of a condition triggering an error attribute will clear the offending error message. [Ref. 4]

The definition of an attribute can be done in different ways: rules defining attributes, rules defining error attributes, and function declarations. [Ref. 4]

An attribute may be declared at the root to define the root environment. This attribute should contain the type bindings of each declared identifier. Subsequent expressions

and subexpressions would the also have an associated type attribute. Typing of every identifier is determined by accessing the root environment while the typing of other expressions is determined by it's associated type attribute. The root environment is propagated from node to node (synthesized) in the tree and local attributes may be defined to allow definition of a computation in one production of a phylum without requiring the computation in all the phyla. [Ref. 4]

An attribute declaration specifies the phylum name, attribute type, and nature of the attribute (synthesized or inherited) for every attribute associated with a phylum. The type of an attribute can be either a predefined phyla or a user defined phylum. Local attributes associate with individual productions instead of with every production of a phylum. Attribute equations associated with each production define the individual attribute values and define how an attribute-definition function is applied to other attribute occurrences of the production to define an attribute occurrence. [Ref. 4]

Upward remote attribute sets are a short cut for repeated inherited attribute definitions and equations. They allow the passing down of values from first corresponding operator up the path of the derivation tree of an expression. The alternative to upward remote attributes sets is to attach

appropriate attributes to each phylum and to write attribute equations to pass the value down the term. [Ref. 4]

A syntactic reference is the use of a part of the term being edited in an attribute definition function. An attribute's type is a phylum that is defined with the same set of rules that are used to define syntactic objects; a program's attribute values and the program itself are all elements of either primitive phyla or phyla defined in the editor specification. Local attribute declarations improve efficiency by allowing the definition of a computation in one production of a phylum without requiring the computation in all the productions. [Ref. 4]

All the error attributes are declared as having type STR, a built-in string phylum of strings. The associated attribute is the null string for no error and the appropriate error message in the case of an error. Each error attribute is defined conditionally, the conditions are checked and the value of the conditional expression is determined by the selected branch of the conditional expression. [Ref. 4]

3. Unparsing Declarations

Unparsing declarations define the display format of a term, which nodes of the abstract syntax tree are selectable, and which productions are editable as text. An unparsing declaration defines the format of a production in terms of strings, sequences of strings, selection symbols, and names of attribute occurrences.

Unparsing rules have two basic forms:

```
phylum : operator [left-side : right-side];
phylum : operator [left-side ::= right-side];
```

The colon indicates that the production should be treated as an indivisible unit and the ::= symbol indicates that text entry is permissible for that production. Different operators of a given phylum may use different symbols. The unparsing declarations for the abbreviated PSDL grammar are shown in Figure 12.

The display of the program on the screen is configured by the following embedded symbols,%t, %b, and %n, designating tabs, reverse tabs, and line feeds respectively. Tab stops are used for lining up items in selective indentation. The tab setting may be changed using the set-parameters command and has a default setting of two.

```
psdl            : PsdlNil       [@ :]
                | PsdlPair      [@ :^["%n"]@]
                ;
component       :EmptyComp       [@ : "<component>"]
                |DaTyComp       [^ : @"<data type>"]
                |OpTyComp       [^ : @"<operator>"]
                ;

data_type       :DaTy           [@: "TYPE" @ @ @    "%n"]
                ;
operator        :Op             [@: "OPERATOR" @ @ @    "%n"]

                ;
```

Figure 12 Grammar Rules Defining Display

Selection symbols @ or ^ designate locations of one of the phylum occurrences in the productions mix-fix representation. The selectability property for a given node in the tree is defined by the selection symbol. The selection symbols define whether a phylum occurrence is a resting place or not. The @ designates a resting place where either of the two corresponding phylum occurrences is specified to be a resting place. The ^ designates a non-resting place, a node where both phylum occurrences are specified as non-resting places. Selections are allowed only at resting places which are determined by the selection symbols of the unparsing declarations. In a generated editor, the cursor is moved to the closest resting place associated with the selected item. The selection symbol @ designates a phylum as a resting place while a ^ does not. A syntax tree node is comprised of left hand side and right hand side phylum occurrences. If either occurrence is specified with an @, then that node is a resting place. [Ref. 4]

4. Concrete Input Syntax

The primary purpose of concrete syntax is that it allows the terms to be entered as text in addition to structure editing as data entry methods. Text entry allows temporary freedom of expression until it is parsed, at which point the string is parsed and translated into an abstract syntax term. [Ref. 4]

The specification of concrete input syntax uses attribute equations which synthesize the terms as attributes of a parse tree in conjunction with concrete input grammar productions to translate text into abstract syntax terms. Every phylum in the abstract syntax that is expressible as text must have a corresponding input syntax phylum. Entry declarations have the forms exhibited in Figure 13.

```
Psdl          {synthesized psdl t; };
Component     {synthesized component t; };
Data_type     {synthesized data_type t; };
Operator      {synthesized operator t; };
Ident         {synthesized identifier t; };
psdl          ~   Psdl.t;
component     ~   Component.t;
data_type     ~   Data_type.t;
operator      ~   Operator.t;
identifier    ~   Ident.t;
```

Figure 13 Association Between Abstract and Input Syntax

The correspondence between the selections within the abstract-syntax tree and entry points within the input syntax is defined using entry declarations in the form:

p ~ P.t;

where the component under the cursor is checked to see if it is a member of p and if it is, its input is parsed to verify that it is a member of phylum P. Following confirmation, the attribute t updates the current selection and the parse tree is abandoned. [Ref. 4]

Lexical phyla, multi-character tokens and keywords, should be defined with individual rules similar to those shown in Figure 14.

```
TYPE:                TypeLex< "type"|"TYPE" >;
OPERATOR:            OperatorLex<"operator"|"OPERATOR">;
SPECIFICATION:  SpecLex<"specification"|SPECIFICATION" >
IDENTIFIER:     IdentifierLex< [a-zA-Z][a-zA-Z0-9_\$]* >;
INTEGER:             IntegerLex< [0-9]+ >;
REAL_NUMERAL:        RealLex<[0-9]+(\.[0-9]+)?
                     ([eE][-+]?[0-9]+)? >;
CHARACTER:      CharacterLex< '.'|'''' >;
STRING:         StringLex<  '(('')|[^'])(('')|[^'])+'  >;
WHITESPACE:          Whitespace<[\ \T\N] >;
```

Figure 14 Translation of Input Syntax into Abstract Syntax

The lexical declarations declare that strings generated by the regular-expressions are in a given phylum. The regular-expressions are enclosed in angle brackets and follow the lex conventions as listed in Figure 15.

c	the character "c"
"$c_1c_2c_3$"	the string "$c_1c_2c_3$"
\c	a "c"
[$c_1c_2c_3$]	the character c_1 or c_2 or c_3
[c_1-c_2]	any of the characters between c_1 and c_2
[^$c_1c_2c_3$]	any character but c_1 or c_2 or c_3
.	any character but newline
^e	an e at the beginning of a line
e$	an e at the end of a line
e?	an optional e
e*	0, 1, 2, ... instances of e
e+	1, 2, 3, ... instances of e
e_1e_2	an e_1 followed by an e_2
e_1|e_2	an e_1 or an e_2
(e)	an e
e_1/e_2	an e_1 but only if followed by e_2
e{n_1,n_2}	n_1 through n_2 occurrences of e.

Figure 15 Regular Expression Notation

The rules for TYPE, OPERATOR, SPECIFICATION and END are examples which define the lexical phyla. The WHITESPACE token identifies blanks and designates them as something to be ignored during parsing. Figure 16 defines the syntax of the concrete input language.

```
Psdl ::=     (Component) {Psdl.t = (Component.t :: PsdlNil); }
        | ( Component Psdl) {Psdl$1.t   =   (Component.t   ::=
        PsdlNil$2.t);} ;
Component ::=(Data_type){Component.t = DaTyComp(Data_type);}
        | (Operator)    {Component.t = DaTyComp(OPerator);} ;
Data_type ::= (Ident Ident Ident)    { D a t a _ t y p e . t    =
        DaTy(Ident$1.t, Ident$2.t, Ident$3.t); };
Operator    ::= (Ident   Ident   Ident)   {Operator.t   =
        Op(Ident$1.t, Ident$2.t, Ident$3.t); };
Ident ::= (IDENTIFIER)  {Ident.t = Identifier(IDENTIFIER); };
```

Figure 16 Syntax of Concrete Input

The parsing declarations define concrete input syntax and are recognizable by their use of ::= to separate the phylum name from the symbols on the right hand side of the declaration. The ability to translate input text into an abstract-syntax tree permits the editor designer to define structural and textual interfaces to the extent desired. [Ref. 4]

5. Transformations and Templates

The ability to transform an object when its structure matches a certain pattern or insert a construct where it is permitted is dictated by transformation declarations. Templates can be inserted for a given placeholder. [Ref. 4]

The form of the transformation declaration is:

```
transform phylum
    on transformation-name pattern : expression;
```

and its use in the PSDL editor is as follows:

```
transform component
    on "data type" <component>:DaTyComp(<data_type>) ,
    on "operator" <component>: OpTyComp(<operator>) ;
```

The transformation is enabled if the current selection matches the named phylum, in this example the phylum "component". The outcome of the transformation is the replacement of "component" with the value of the expression on the right of the ":" with the selected expression for the data type or the operator. [Ref. 4]

E. EVOLUTION

The evolution of the PSDL syntax directed editor initially should entail the continuation of this process, that is, the refinement of the constructs of the language to encompass the complete description of the language and the semantics within the language. PSDL type-checking, inheritance constraints, and stream-type calculations should be checked for consistency and inheritance by the assignment of attribute equations. As CAPS evolves and further refinements to the language occur, the language naturally will need to be upgraded, and the Cornell Synthesizer Generator may be applied to other aspects of the CAPS system including debugging, scheduling, and translation.

V. USER GUIDE TO THE PROTOTYPING LANGUAGE EDITOR

A. GENERIC FEATURES OF THE EDITOR

The top line of the screen (see Figure 17) has a highlighted title bar displaying the name of the current buffer. The remainder of the screen is divided into three regions: the command line, the object pane, and the help pane. [Ref. 4]

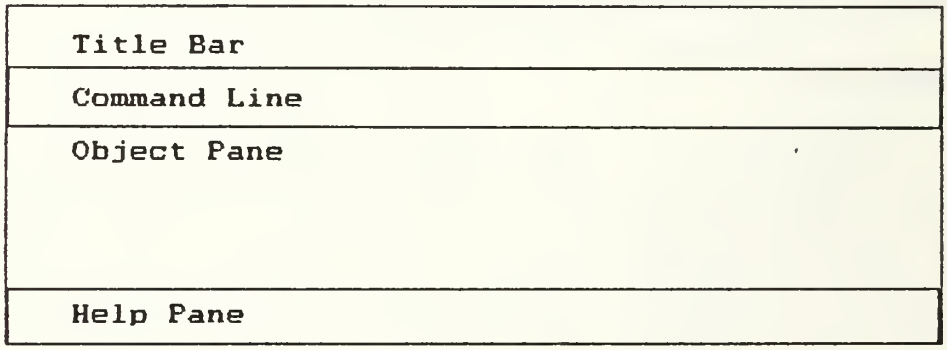

Figure 17 Editing Session Screen

The command line echoes commands and displays system messages, the object pane displays the buffer's program segment, and the help pane shows which constituent is selected. [Ref. 4]

The editor is a screen-oriented hybrid editor. The program being edited is an object which has a hierarchical structure. The object being edited is built through the use of templates, placeholders, and text. Templates are formatted patterns of keywords and placeholders. Placeholders identify legal locations where new components can be added. Text editing is performed at the character selection, denoted by the I-beam symbol within the highlighted section. Both the structural selection and the character selector are keyboard or mouse controlled. Using the mouse, the arrow selector is first positioned, then it is selected using the left mouse button. Text editing is not permitted in all constructs and the editor will display an error message if attempts are made to insert text in other than character selection text buffers. Text input is checked by the parser after entry for syntactic correctness before being accepted as a valid program input. Before proceeding with the program, erroneous text must be either corrected or deleted. [Ref. 4]

The Prototyping Language Editor places objects into a collection of named buffers. Normally, each edited file has a unique buffer. The objects contained in buffers are called terms which are derivation trees with respect to the abstract syntax of the language. The nodes of a term are instances of operators and the subtrees of a node are the operator's

arguments, themselves terms. Each term has a textual output representation. [Ref. 4]

Each buffer has a selection, the area of interest to be edited. The selection can be chosen by tree walking commands such as forward-preorder, backward-preorder, forward-sibling, backward-sibling, and ascend-to-parent. [Ref. 4]

The objects edited are stored in named buffers. A file being edited normally is placed in its own buffer. That buffer remains bound to the file until the buffer is exited or a different file is explicitly read into that buffer. [Ref. 4]

Every buffer has a syntactic mode which is declared in the root phylum. This mode is automatically maintained by the editor and the value of the buffer remains syntactically well formed. [Ref. 7]

A buffer of a syntactic mode always has a completing term corresponding to its phylum. During editing, a buffer generally contains a term with several placeholders. As editing proceeds, the buffer is updated by replacing constituent subterms in any order. Placeholders are replaced by terms created by text entry, transformation, template insertion, or cutting and pasting. [Ref. 7]

The mouse can be used to point anywhere on the object pane, and click to a new selection. The mouse can be used in conjunction with the select-start/select-stop commands to drag between characters in a given production. [Ref. 10]

The mouse also actuates menus for the various commands as shown in Figure 18. [Ref. 7]

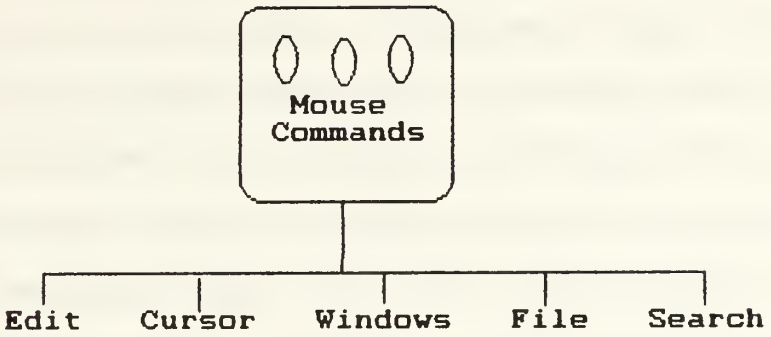

Figure 18 Mouse Commands

The middle button activates a menu with the following commands: edit, cursor, windows, file, and search. [Ref. 10]

Each editing transaction replaces a subterm or sublist. A transformation determines a replacement value for the selected subterm as a function of its current value. Transformations are enabled or disabled in accordance with the matching of its pattern with that of the selection. Enabled transformations are substitutions that can be made for certain values, for example, <component> could be replaced by ''<data_type>'' or ''<operator>''. Transformations cannot introduce context-free syntax errors since their definitions are also type-checked when the Synthesizer Specification Language (SSL) specification is compiled into an editor. [Ref. 4]

The Synthesizer Generator editors have a language dependent part and a set of system commands which are

language independent and strictly control editor functions, for example cutting and pasting. To ensure connectivity of the tree structure, the editor inserts placeholders to replace any deleted subterms. [Ref. 4]

The textual display of the selection can be edited like a typical screen-oriented text editor. The text, initially is placed into a text buffer upon the first action that implies textual modification. The text buffer is then displayed on the screen instead of the selection, causing its existence to be practically invisible to the user. Within the text buffer, editing is unrestricted. A character selection, identifying the location that text changes occur, can be positioned by moving the cursor or the mouse. When the term selection is moved away from its current location, the text is parsed with respect to the concrete input syntax of the current selection context. If a syntax error is detected it must be corrected before the contents of the text buffer can be translated into a term, which in turn, replaces the original subterm and is displayed according to its output representation. The generated effect may be to format the text on the screen or transform it into an equivalent representation. [Ref. 4]

A term in a buffer is attributed, which means it has associated with it computed values that characterize the term. Every time a buffer is modified, the attributes are revised so that they remain current and form a consistent

database of derived information. Selected attribute values are displayed to the user as part of the output presentation during the editing session. [Ref. 4]

The buffer contents can be written into one of two types of files, an abstract structure file which records the abstract structure of the terms, or a textual file which records the textual output representation. Generation of a text file is possible only if the concrete input syntax is complete. [Ref. 4]

B. LANGUAGE SPECIFIC FEATURES OF THE EDITOR

This implementation of a PSDL editor is rudimentary in that it is so abbreviated as to be ineffectual for the PSDL programmer, but it demonstrates the feasibility of the Synthesizer Generator for follow-on work. The current version of the editor limits the PSDL to components which may be expressed only as operators and data types. The language specific details of the editor are contained in file psdl.ssl which are listed in Appendix C. The remainder of this guide assumes a fully implemented editor.

C. USING THE EDITOR

The Prototyping Language editor is invoked by typing psdl.syn with or without parameters. The parameter list specifies a collection of named files to be loaded into editor buffers.

The command to exit the PSDL editor is ^C or the exit command found in the file menu using the mouse. Commands can be invoked by direct keyboard entry, or selected from a menu. Keys that are not bound to editor commands show up as text on the object being edited.

The Prototyping Language Editor applies knowledge of PSDL's context-free syntax to assure that programs are always syntactically well formed. The PSDL code is represented within the editor by a derivation tree with respect to the underlying context-free PSDL grammar. The program is modified by two mechanisms: structural editing and textual editing. [Ref. 4]

1. Structural Editing

Structural editing treats the program as a hierarchy of computational modules. Programs are created in a top-down manner using predefined formatted language constructs known as templates. An example of a template is the interface construct:

<attribute>

BY REQUIREMENTS <identifier>

where <attribute> and <identifier> are placeholders that show where additional program definition and refinement may be entered. Programs are developed by inserting new templates into placeholders of entered templates, for example, any of the following templates could be inserted into attribute: INPUT <type_decl>, OUTPUT <type_decl>, STATES <type_decl>

INITIALLY <expression>, EXCEPTIONS <id_list>, or MAXIMUM EXECUTION TIME <time>. Placeholders in templates prompt the user for legal insertions and enforce the syntactic constraints. [Ref. 4]

The cursor points to the current selection, the point where insertion and deletion may be performed. The selection can be moved from one template to another and from one template to its component parts. [Ref. 4]

Legal templates may be inserted into a program and textual entries may be added after the editor verifies the legality of the insertion. For example, if the selection is <attribute>, a menu command to insert a state attribute into the program would result in the following properly indented program fragment:

```
STATES <type_decl>
    INITIALLY <expression>
        BY REQUIREMENTS <identifier>;
```

notice that by the recursive nature of the language, further definition can continue indefinitely. The menu of insertion commands does not provide the same choices in all contexts, it is limited to the legal choices for that point in the program. In The Synthesizer Generator [Ref. 4] Reps and Teitelbaum state:

> Templates eliminate mundane tasks of program development and let the programmer focus on the intellectually challenging aspects of programming. Each template insertion is syntactically correct because template commands are valid only in appropriate contexts. Indentation is automatic: both when a

> template is introduced and when it is moved. Typographical errors in structural units are impossible; the templates are predefined and immutable, so after a template has been inserted, errors cannot be introduced by subsequently modifying it. Thus, a program developed with a structure editor is always well formed, regardless of whether it is complete.
>
> Templates correspond to abstract computational units. Because they are inserted and manipulated as units, the process of programming begins and continues at a high level of abstraction. [Ref. 4]

Transformation operations allow the replacement of one piece of code with another within legal constraints. An example of a transformation could be the transformation of time units into a standardized unit say ms, so that .001 sec would be transformed into 1 ms. [Ref. 4]

2. Textual Editing

The Prototyping Language Editor is a hybrid editor which combines the advantages of a structure editor with text editing capabilities. Syntactically incorrect programs are prevented by validation of the code (parsing of text) before incorporating into the structured derivation tree. [Ref. 4]

The textual component of the Prototyping Language Editor allows the composition of code as text, and the editing of existing code as text. [Ref. 4]

3. Demonstration of Prototyping Language Editor

This demonstration illustrates pretty printing, list manipulation and text editing. Initially, the screen appears with the following information in the indicated regions. The title bar at the top line of the screen displays the name of the current buffer. The remainder of

the screen is divided into the three regions shown in Figure 17. The command line echoes commands and lists system messages, currently blank. The help pane provides information about the current selection and its allowable transformations, in this case:

Positioned at component data_type operator.

The object pane displays part of the program being edited, <component>. [Ref. 4]

Invoking a transformation in the editor lists the initial choices of data_type or operator. Selecting the data type displays the following in the object pane:

TYPE <identifier>

<type specification>

<type implementation>.

Likewise, selecting the operator displays the following template:

OPERATOR <identifier>

<operator specification>

<operator implementation>. [Ref. 4]

Since the editor is a hybrid editor supporting both structure editing and textual entry, the method of delineating which form of editing must be distinguishable. In structure editing, the structural selection is marked by the highlighting of the applicable template or placeholder, while an I beam is used as a character selector in text editable portions. The structural selection and the

character selection can be positioned on the screen using the mouse or the positioning commands:

ESC-p for cursor up

ESC-d for cursor down

ESC-b for cursor left

ESC-f for cursor right.

The select command is bound to ESC-@. A complete set of editor commands is contained in Appendix F. [Ref. 4]

Placeholders can be manipulated directly as text or by template insertion. Available templates are listed in the help pane and are addressable by clicking on the name in the help pane, through the mouse driven pop-up menus, and by the keyboard using the execute-command (Tab (^I) or ESC-x). To illustrate this, consider the interface template in the object pane:

<attribute> <reqts_trace>

when the cursor is positioned over the <attribute> the help pane displays the following:

Positioned at attribute generic_param input output states exceptions timing_info

informing the user that the templates available for insertion are only those listed. Likewise the transformation pop-up menu would exhibit the same information in a different format or the selection could be typed in using the execute-command. When a template is selected, say input, the command line echoes: COMMAND: input. [Ref. 4]

Template commands can be terminated by moving the buffer selection. One of the most natural of these commands is the return (^M) which causes the selection to move to the next resting place (actually a preorder traversal of the derivation tree). [Ref. 4]

When the placeholder represents a list or an optional list, a forward-with-optionals command (return (^M)), moves the selection to the next element in the list. To terminate the list, another forward-with-optionals command (return (^M)) will advance to the next template or placeholder. An example of a list is an identifier list:

dog, cat, whale, wombat, <id_list>. [Ref. 4]

A template may have more than one placeholder, for example type_decl, where there is a left-hand-side id_list and a right-hand-side type_name. When a template is inserted, a forward_with_optionals command is invoked causing the selection to move to the first placeholder, id_list. After the id_list is inserted, two returns (forward_with _optionals, (^M)) cause the text to be parsed and if syntactically correct, the selection moves to the right-hand side of the type declaration, type_name. [Ref. 4]

Some selections are editable as text in which case the selection is captured into a text buffer. The text buffer is located at the selection and is not distinguishable as a text buffer until a character is entered at which point the character cursor appears, represented as an I beam.

While the text is being edited, the selection exists as text, not as structure and operations within the selection are defined only on characters, not on program structures. Termination of text entry into the text buffer is signalled by a return (^M forward-with-optionals). Since the user types in the text buffer, it is possible to contain errors. To prevent the introduction of errors into the program, a parser checks the buffer and if a syntax error exists, sounds a warning signal, displays an error message on the command line, and positions the cursor at the end of the word containing the first error. Following the detection of an error, text editing may be resumed to correct the error. To erase the selection, the delete_section (^K) can be invoked and the selection re-entered. After the correction, return (forward-with-optionals) causes the text to be parsed again. [Ref. 4]

Syntactic correctness allows the parsing, but does not guarantee that the editor will accept the input without complaint; a comment revealing failure to comply with data types or control constraints will be generated if the attributes are not satisfied. [Ref. 4]

This concludes a brief introduction to the Prototyping Language Editor. A complete set of commands is listed in Appendix F and more complete details can be found in The Synthesizer Generator Reference Manual [Ref. 10].

VI. CONCLUSIONS AND FOLLOW-ON WORK

This thesis has demonstrated the feasibility of the software engineering approach designed by Dr. Berzins and Dr. Luqi as a method of breaking a complex problem into simpler pieces both from the point of view of CAPS as a system and also from the perspective of designing a language specific editor using the Cornell Synthesizer Generator. Much of the ground work has been laid in the sense that a restricted PSDL editor, psdl.syn, has been generated which can provide the basis for comprehensive expansion of the Prototyping System Design Language features.

Whereas progress has been made in identifying the nature of the problem of specifying the language-dependent features of a language-based editor and the theory and documentation are more clearly defined, there remains significant additional implementation programming to transform the PSDL editor into a tool commensurate with the remainder of the CAPS system. The Abstract Syntax, Appendix B, the Unparsing Declarations, Appendix D, and the Concrete Input Syntax, Appendix E, are largely written, but untested. The work remaining to complete these parts, notably in the Unparsing Declarations, the attribute equations, fitting together the modules, and testing and debugging the system is a significant task. Due to the simplicity of the interface

with the remainder of the CAPS system it is not anticipated that the interface will cause significant problems. The most likely problem will be the ability of the PSDL editor to accept the PSDL code generated by the Graphics Editor. This problem can be resolved by the unparsing declarations and the lexical analysis. The abstract syntax should be refined and the approach to declaring items optional should be streamlined and simplified.

There is also application for the use of the Cornell Synthesizer Generator in other portions of the CAPS system. The Synthesizer Generator has great power in its ability to transform input into various forms and is quite capable in the areas of consistency checking and verification of conventions. It could be adapted to assist in the tasks of translation, scheduling, data base keying, and debugging.

APPENDIX A: BACKUS-NAUR FORM FOR PSDL

Optional items are enclosed in [square brackets] and items that may appear zero or more times appear in <braces>* Terminal symbols appear in " double quotes ."

```
<psdl> ::=   <component>*

<component> ::=   data_type
                  | operator

<data_type> ::=   "type" id type_spec type_impl

<operator> ::=    "operator" id operator_spec operator_impl

<type_spec> ::=   "specification" [type_decl]
                  <"operator" id operator _spec>*
                  [functionality] "end"

<operator_spec> ::=    "specification" interface
                       [functionality] "end"

<interface> ::=   <attribute [reqmts_trace]>*

<attribute> ::=   generic_param
                  | input
                  | output
                  | states
                  | exceptions
                  | timing_info

<generic_param> ::=    "generic" type_decl

<input> ::=  "input" type_decl

<output> ::= "output" type-decl

<states> ::= "states" type-decl "initially" expression_list

<exceptions> ::=  "exception" id_list

<id_list> ::=          <id_list - id <"," id>*
```

```
<timing_infor> ::=     ["maximum execution time" time]
                       ["minimum calling period" time]
                       ["maximum response time" time]

<time> ::=   integer [unit]

<unit> ::=   "microsec"
             | "ms"
             | "sec"
             | "min"
             | "hours"

<reqmts_trace> ::=     "by requirements" id_list

<functionality> ::=    [keywords]
                       [informal_desc]
                       [formal_desc]

<keywords> ::=    "keywords" id_list

<informal_desc> ::=    "description" " <" text ">* "

<formal_desc> ::= "axioms" " <" text ">* "

<type_impl> ::=   "implementation" "Ada" id
                  | "implementation" type_name
                  <"operator" id operator_impl>* "end"

<operator_impl> ::=    "implementation" "Ada" id
                       | "implementation" psdl_impl

<psdl_impl> ::=   data_flow_diagram
                  [streams]
                  [timers]
                  [control_constraints]
                  [informal_desc]
                 "end"

<data_flow_diagram> ::=     "graph" <link>*

<link> ::=   id "." id [":" time] "->" id

<streams> ::=     "data stream" type_decl

<type_decl> ::=   id_list  ":"  type_name  <","  id_list  ":"
                  type_name>*

<type_name> ::=   id " [" type_decl "] "
                  | id

<timers> ::= "timer" id_list
```

```
<control_constraints> ::=   "control constraints"
                            <constraint>*"

<constraint> ::=  "operator" id
                  ["triggered" [trigger] ["if" predicate]
                   [reqmts_trace]]
                  ["period" time [reqmts_trace]]
                  ["finish within" time [reqmts_trace]]
                  <"output" id_list "if" predicate
                    [reqmts_trace]>*
                  <"exception" id ["if" predicate]
                    [reqmts_trace]>*
                  <timer_op id ["if" predicate]
                   [reqmts_trace]>*

<timer_op> ::=    "RESET timer"
                  | "START timer"
                  | "STOP timer"
                  | "READ timer"

<trigger> ::=     "by all" id_list
            | "by some" id_list

<predicate> ::=   "not" predicate
                  | predicate "and" predicate
                  | predicate "or" predicate
                  | expression
                  | id ":" id_list

<relation> ::=    simple_expression
                  | simple_expression rel_op
simple_expression

<simple_expression> ::=     [sign] integer [unit]
                        | [sign] real
                        | ["not"] id
                        | string
                        | ["not"] "(" predicate ")"
                        | ["not"] boolean constant

<rel_op> ::= "<"
             | "<="
             | ">"
             | ">="
             | "="
             | "=/"
             | ":"

<expression> ::=  constant
                  | id
                  | type_name "." id "(" expression_list ")"
```

```
<expression_list> ::=   expression <"," expression>*
*****
<real> ::=   integer "." integer

<sign> ::=   "+"
             | "-"


<bool_op> ::=     "and"
                  | "or"

<boolean_constant> ::= "true"
                       | "false"

<string> ::= """ <char>*"""

<text> ::=   """ <char>*"""
```

APPENDIX B: COMPLETE PSDL ABSTRACT SYNTAX

```
root psdl;      /* tree root */

optional list psdl;
psdl    : PsdlNil() /* This is a completing term */
        |PsdlPair(component psdl)
            /* allows recursive addition of components
            within the program */
        ;

component: EmptyComp()  /* required as a completing term */
    | DaTyComp(data_type)   /* function calls */
    | OpTyComp(operator)
    ;

data_type: DaTy(id type_spec operator_spec) /*  notice  that
                            the keyword "type" will have to
                            be  inserted  in  the  actual  text
                            of the output program */

operator: Op(id operator_spec op_impl)  /* notice that the
                keyword "operator" will have to be
                inserted in the actual text of the
                output program */

/* Things begin to get a bit tricky here, notice that within
what should be a function are items which are optional here
i.e., type_decl, though not necessarily optional everywhere
i.e., type_decl in generic_param , fortunately, functionality
appears always to be optional and can be defined as optional
- consider defining an optional function */

type_spec       : TySpec(optional_type_decl
            operator_specList
            functionality)
            ;   /* notice that there is no
                optional_type_decl in psdl  grammar.
                Also note that the keywords
                "specification","operator" and "end"
                will have to be inserted in the actual
                text of the output program */

optional list operator_specList;    /* the optional makes it
                                    a list of zero or more
                                    vice one or more*/
```

```
operator_specList:  Op_SpecListNil()
            | Op_SpecListPair(operator_spec_clause
                        operator_specList)
            ;   /* operator_spec_clause does not exist
                in psdl*/

operator_spec_clause: OpSpecClause( id operator_spec )

optional optional_type_decl ;
optional_type_decl :    OptTDeclNull()
            | OptTDecl(type_decl)
            ;   /* modeled from sec 5.2 in The
                Synthesizer Generator, not absolutely
                sure that this whole optional fuss
                isn't avoidable by declaring
                type-decl to be optional in the
                type_spec declaration */

operator_spec   : OpSpec    ( interface
                functionality)
                ;   /* notice that the keywords
                "specification" and "end" will have to
                be inserted in the actual text of the
                output program */

optional list interfaceList;
interfaceList   :   IfaceListNil()
            | IfaceListPair(interface interfaceList)
            ;

interface   :   Iface(attribute reqmts_trace)
            ;   /* reqmts_trace is always optional */

attribute   :   EmptyAttr()
        |   AttrGenParam(generic_parameter)
        |   AttrInput(input)
        |   AttrOutput(output)
        |   AttrStates(states)
        |   AttrExceptions(exceptions)
        |   AttrTimgInfo(timing_info)
        ;

generic_param   :   GenParam(type_decl)
                ;   /* still need keyword "generic"*/

input       :   Input(type_decl)    /* still need keyword
                        "input"*/
            ;
```

```
output      :   Output(type_decl)   /* still need keyword
                        "output"*/
            ;

states      :   States(type_decl expression_list)
            ;   /* still need keywords "states" and
                "initially"*/

list expression_list;
expression_list :   ExprListNil()
        |   ExprListPair(expression expression_list)
        ;

exceptions  :   Exceptions(id_list)
            ;   /* still need keyword "exceptions"*/

list id_list    ;
id_list     :   IdListNil()
            |   IdListPair(id id_list)
            ;   /* This is written up in the grammar as a
mandatory entry followed by an optional list separated by
commas. This is represented as an ordinary list (one or more)
- may have to redefine in order to be able to insert commas
(commas OK, taken care of in unparsing declarations */


optional timing_information;
timing_information  :   TimeInfoNull()
                |   TimeInfo( max_execution
                    min_call_period
                    max_response)
                ;

optional max_execution;
max_execution   :MaxExecutionNull()
            MaxExecution(time)
            ;

optional min_call_period;
min_call_period :MinCallPeriodNull()
            MinCallPeriod(time)
            ;

optional max_response;
max_response    :MaxResponseNull()
            MaxResponse(time)
            ;   /* whereas the command is identical, there
are three keyword sequences possible: "maximum execution
time", "minimum calling period", "maximum response time"-
May have distorted meaning of the language: since time is
```

```
already optional, this implies that could leave either of
these three in the specification without entering a time */

optional time;  /*  need  to  verify  that  time  is  always
optional - not strictly optional in timing_info*/

time    :   TimeNull()
        | Time(integer unit)
        ;   /* notice that unit needs to be defined as
            optional*/

optional unit;
unit    :   EmptyUnit()
        | Us()
        | Ms()
        | Sec()
        | Min()
        | Hours()
        ;   /* unit selects the units of time in "microsec",
"ms", "sec", "min", or hours ; this doesn't look right to me,
it seems like there ought to be a scaling factor */

optional reqmts_trace;
reqmts_trace:   ReqmtsTraceNull()
        | ReqmtsTrace(id_list)
        ;/* not defined in psdl as optional, but always used
        optionally (this allows easier definition of phyla
        that use requirements_trace)*/

optional functionality
/* if functionality is always optional and everything in
functionality  is  optional,  then  the  things  defining
functionality would be doubly optional- but since it appears
that the elements can be strung one after the other each
should still be defined as optional */

functionality   :   FunctionalityNull()
                    Functionality   (optional_keyword
                    optional_informal_desc
                    optional_formal_desc)
                ;

optional optional_keyword;
optional_keyword:   OptKeywdNull()
            | OptKeywd(keyword)
            ;

optional optional_informal_desc;
optional_informal_desc: OptInfDescNull()
            | OptInfDesc(informal_desc)
```

```
optional optional_formal_desc;
optional_formal_desc:   OptFormalDescNull()
            | OptFormalDesc(formal_desc)
            ;

keywords:   KeyWords(id_list)
        ;   /* missing keyword "keywords"*/

informal_desc:  InfDesc(text)
        ;   /*  missing  keywords  "description",  "{",  and
"}"*/

formal_desc:    FormalDesc(text)
        ;   /* missing keywords "axioms", "{", and "}"*/

/* recheck to see if must start with an empty option (look at

attribute example)*/
type_impl:  AdaTyImpl(id)   /* missing keywords
            "implementation" and "Ada"*/
        | TyImpl(type_name optional_operator_impl)
        ;   /* optional-operator impl is not a psdl
            grammar type.  Also missing keywords
            "implementation" and "end"*/

optional list optional_operator_impl;
optional_operator_impl: OptOpImplNull()
            | OptOpImpl(id operator_impl)
            ;   /* missing keyword "operator"*/

operator_impl:  AdaOpImpl(id)   /* missing keywords
                "implementation", "Ada" */
            | PsdlOpImpl(psdl_impl)
            ;       /* missing keyword "implementation"*/

psdl_impl:  PsdlImpl(data_flow_diagram
            optional_streams
            optional_timers
            optional_control_constraints
            optional_informal_desc)
            ;   /* missing keyword "end"*/

optional optional_streams;
optional_streams:   OptStreamsNull()
            | OptStreams(streams)
            ;
```

```
optional optional_timers;
optional_timers:    OptTimersNull()
            | OptTimers(timers)
            ;

optional optional_control_constraints;
optional_control_constraints:OptConConstrNull()
            | OptConConstr(control_constraints)
            ;

optional list data_flow_diagramList ;
data_flow_diagramList:  DaFlDiagramNil()
            | DaFlDiagramPair(data_flow_diagram
                        data_flow_diagramList)
            ;

data_flow_diagram:  DaFlDiagram(link)
            ; /* need keyword "graph"*/

link :          Link(id operator_id id)
            ;   /* need keywords "." and "->"*/

operator_id:            Id(id time)
            ;   /* need keyword ":" Notice that time
is defined as an optional phylum */

streams:        Streams(type_decl)
            ;   /* Need keywords "data stream" */

optional list type_declList;
type_declList:      TyDeclListNil()
            | TyDeclListPair(type_decl type_declList)
            ;
            /* this is not defined specifically in
            psdl, but type decl is always used optionally
            and it is defined as a list separated by
            commas (like id_list).  Notice that keyword
            "," needs to be added*/

type_decl:      TyDecl(id_list type_name)
            ;   /* keyword ":" needs to be added*/

type_name   :   EmptyTyName()
        |   TyNameId(id)
        |   TyNameTyId(id type_decl)
        ;   /* remember type is always used optionally and
            is declared to be optional*/

timers      :   Timers(id_list)
        ;   /* missing keyword "timer"*/
```

```
optional list control_constraintsList;
control_constraintsList:ConConstrListNil()
            | ConConstrListPair(control_constraints
            control_constraintsList)
            ;   /* missing keyword "control
                constraints"*/

/* this construction has a list of optional and nested
optional constructs; for the most part, the constructs
themselves are always optional so will be defined as optional
the optional_name method will only be used when the optional
phylum has mandatory instances*/

constraint  :   Constraint(id trigger optional_predicate
            reqmts_trace time reqmts_trace time reqmts_trace
            optional_id_listList
            optional_exception_listList
            optional_timer_opList)
            ;   /* the last three terms are defined as
                optional lists*/

optional optional_predicate ;
optional_predicate :    OptPredicateNull()
            | OptPredicate(predicate)
            ;   /* modeled from sec 5.2 in The
                Synthesizer Generator */
optional list optional_id_listList;
optional_id_listList:   OptIdListNil()
            | OptIdListPair(optional_id_list
            optional_id_listList)
            ;   /* this represents {id_list predicate
                [reqmts_trace]}*/

optional_id_list:   OptIdList(id_list predicate reqmts_trace)

            ; /* reqmts_trace is defined to be optional*/

optional list optional_exception_listList;
optional_exception_listList:    OptExcepListNil()
            | OptExcepListPair(optional_exception_list
            optional_exception_listList)
            ;   /* this represents {id [predicate]
                [reqmts_trace]}*/

optional_exception_list:    OptExcepList(id
optional_predicate reqmts_trace)
            ; /* reqmts_trace is defined to be optional*/
```

```
optional list optional_timer_opList;
optional_timer_opList:  OptTiOpListNil()
            | OptTiOpListPair(optional_timer_op
            optional_timer_opList)
            ;   /* this represents {id_list predicate
                [reqmts_trace]}*/

optional_timer_op:  OptTiOp(timer_op  id  optional_predicate
reqmts_trace)
            ; /* reqmts_trace is defined to be optional*/


timer_op    :   EmptyTimer_Op()
        |   ReadTimer()
        |   ResetTimer()
        |   StartTimer()
        |   StopTimer()
        ;   /*  timer_op  determines  which  timer  operation
occurs i.e., "RESET timer", "START timer", "STOP timer", or
"READ timer"; hopefully there can be assigned some control
associated with this command */

optional trigger ;
trigger     :   TriggerNull()
            | ByAllTrigger(identifier_list)
            | BySomeTrigger(identifier_list)
            ;   /* notice that this is an or construct but
the only difference is in the keywords "by all" or "by some"
*/

predicate   :   EmptyPred()
        |   NotPred(predicate)
        |   AndPred(predicate predicate)
        |   OrPred(predicate predicate)
        |   Expression(expression)
        |   Id(id id_list)
        ;

expression  :   EmptyExpression()
        |   ConstExpr(constant)
        |   ExprId(id)
        |   TyExpr(type_name id expression_list)
        ;

constant    :   EmptyConst()
        |   Number(numeric_constant)
        |   Boolean(boolean_constant)
        ;
```

```
numeric_constant:   EmptyNumber()
        |   Real(REAL)
        |   Integer(INTEGER)
        ;

boolean_constant:   EmptyBool()
        |   True()
        |   False()
        ;

list expressionList;    /* the optional makes it a
                list of zero or more vice one
                or more*/
expressionList  :   ExpressionListNil()
            | ExpressionListPair( expression
                        expressionList)
            ;
```

APPENDIX C: PSDL.SSL CURRENT PARTIAL IMPLEMENTATION

```
root psdl;          /* tree root */

optional list psdl;
psdl : PsdlNil()    /* This is a completing term */
     |PsdlPair(component psdl)
               /* allows recursive addition of components
               within the program */
     ;

component: EmptyComp()    /* required as a completing term */

     | DaTyComp(data_type)     /* function calls */
     | OpTyComp(operator)
     ;


data_type: DaTy(identifier identifier identifier);
operator: Op(identifier identifier identifier)
                /* notice that the
                    keyword "operator" will have to be
                    inserted in the actual text of the
                    output program */
            ;

identifier: IdentifierNull()
        | Identifer(IDENTIFIER)
        ;

                    /* UNPARSING DECLARATIONS */

psdl            : PsdlNil       [@ :]
                | PsdlPair      [@ :^["%n"]@]
                ;

component       :EmptyComp      [@ : "<component>"]
                |DaTyComp       [^ : @"<data type>"]
                |OpTyComp       [^ : @"<operator>"]
                ;                    /* try ^'s as @'s during
refinement*/

data_type       :DaTy           [@: "TYPE" @ @ @    "%n"]
                ;
```

```
operator        :Op             [@: "OPERATOR" @ @ @     "%n"]
                ;


TYPE:           TypeLex< "type"|"TYPE" >;
OPERATOR:       OperatorLex< "OPERATOR" >;
IDENTIFIER: IdentLex<[a-zA-Z][a-zA-Z0-9]*|[?] >;
WHITESPACE:      Whitespace< [\ \t\n]* >;
INTEGER:    IntegerLex< [0-9]+ >;
REAL_NUMERAL:    RealLex<  [0-9]+(\.[0-9]+)?([eE][-+]?[0-9]+)?
>;
CHARACTER:      CharacterLex< '.'|'''' >;
STRING:         StringLex< '((')|[^'])((')|[^'])+' >;

Psdl             {synthesized psdl t; };
Component        {synthesized component t; };
Data_type        {synthesized data_type t; };
Operator         {synthesized operator t; };
Ident       {synthesized identifier t; };
psdl         ~   Psdl.t;
component    ~   Component.t;
data_type    ~   Data_type.t;
operator     ~   Operator.t;
identifier~ Ident.t;
Psdl ::=    (Component) {Psdl.t = (Component.t :: PsdlNil); }
    | ( Component Psdl) {Psdl$1.t    =    (Component.t    ::
PsdlNil$2.t);} ;
Component ::=(Data_type){Component.t = DaTyComp(Data_type);}
    | (Operator)    {Component.t = DaTyComp(OPerator);} ;
Data_type ::= (Ident Ident Ident)    { D a t a _ t y p e . t     =
DaTy(Ident$1.t, Ident$2.t, Ident$3.t); };
Operator    ::= (Ident    Ident    Ident)    {Operator.t    =
Op(Ident$1.t, Ident$2.t, Ident$3.t); };
Ident ::= (IDENTIFIER)  {Ident.t = Identifier(IDENTIFIER); };
transform component
    on "data type" <component>:DaTyComp(<data_type>) ,
    on "operator" <component>: OpTyComp(<operator>) ;
```

APPENDIX D: UNPARSING DECLARATIONS

```
psdl            : PsdlNil       [@ :]
                : PsdlPair  [@ :^[%n]@]
                ;

component       :EmptyComp  [@ : "<component>"]
                |DaTyComp       [^ : @"<data type>"]
                |OpTyComp       [^ : @"<operator>"]
                ;

data_type       :DaTy       [@: "TYPE" @ @ @    %n]
                ;

operator        :Op         [@: "OPERATOR" @ @ @    %n]
                ;

type_spec       :TySpec     [@:"SPECIFICATION" @    %n%t
/* how to list an optional keyword*/
                        "OPERATOR" @ @          %n%t
                        @                   %n
                        "END"]
                        ;
operator_specList:Op_SpecListNil[@::=]
            |OpSpecListPair[@ :^[%n]@]
            ;

operator_spec_clause: OpSpecClause[^ : ^ ^]
            ;

optional_type_decl :    OptTDeclNull[@    ::=    "<type
                        declaration>"]
                    | OptTDecl  [^ : ^ ]
                    ;

operator_spec   : OpSpec        [@ : "SPECIFICATION" @  %t%n
                            @               %n
                            @               %b%n
                        "END"]
                ;

interfaceList   : IfaceListNil      [@ :]
                | IfaceListPair [@ :^[%n]@]
                ;

interface   :   Iface           [^ ::= ^ ^];
```

```
attribute    :   EmptyAttr [^ ::=]
             |   AttrGenParam [^ ::= ^]
             |   AttrInput [^ ::= ^]
             |   AttrOutput [^ ::= ^]
             |   AttrStates [^ ::= ^]
             |   AttrExceptions [^ ::= ^]
             |   AttrTimgInfo [^ ::= ^]
             ;

generic_param    :   GenParam [@ : "GENERIC" @]
                 ;

input        :   Input [ @ :"INPUT" @ ]
             /* DIDN'T USE, NOT SUPPORTED BY GRAMMAR ["%n"] @]
             non comma separated list sample p 60  TSG Ref
             Man. This follows Janson App B example, not App A
             grammar.  PSDL should support a list of inputs
             need to check and see if defined elsewhere or if
             input should be defined as a list */
             ;

output       :   Output [ @ : "OUTPUT" @ ]
             /* ditto previous (input) comment */
             ;

states       :   States [@ : "STATES" @      "%t%n"
                 "INITIALLY" @]
             ;

expression_list :   ExprListNil [@ ::=]
                |   ExprListPair[@ ::= ^ [%n] @]
                ;

exceptions   :   Exceptions [^ : "EXCEPTIONS" @]
             /* notice that second @ is a list          */
             ;

list id_list     ;
id_list          :   IdListNil [@:]
                 |   IdListPair [@: ^ [","%n] @]
                 ;

timing_information  :TimeInfoNull    [@    ::=    "<timing
information>"
                    |TimeInfo    [^ ::= ^ ^ ^]
                    ;
```

```
max_execution   :MaxExecutionNull   [@    : : =    "<maximum
                                    execution time>"]
                 MaxExecution       [^ ::= "MAXIMUM EXECUTION
                                    TIME" ^]
                ;


min_call_period :MinCallPeriodNull  [@ ::= "<minimum calling
                                           period>"]
                MinCallPeriod   [^   ::=   "MINIMUM   CALLING
                                           PERIOD" ^]
                ;

max_response    :MaxResponseNull    [@ ::= "<maximum response
                                           time>"]
            MaxResponse     [^ ::= "MAXIMUM RESPONSE TIME" ^]
            ;

optional time;
time    : TimeNull  [@ ::= "<time>"]
        | Time  [^ ::= ^ ^]
        ;

unit    :   EmptyUnit   [^ ::= "<unit>"]
        |   Us      [@ : "microseconds"]
        |   Ms      [@ : "ms"]
        |   Sec     [@ : "sec"]
        |   Min     [@ : "min"]
        |  Hours    [@ : "hours"]
        ;

reqmts_trace:   ReqmtsTraceNull [@ ::= "<requirements>"]
            | ReqmtsTrace   [^ ::= "BY REQUIREMENTS" ^]
            ;

functionality   :   FunctionalityNull   [   @        :   :   =
                                        "<functionality>"]
                    Functionality   [^ ::= ^ ^ ^]
                ;

optional_keyword:   OptKeywdNull    [@ ::= "<keywords>"]
                | OptKeywd [^ ::= ^]
                ;

optional_informal_desc: OptInfDescNull  [@   ::=   "<informal
description>"]
                      | OptInfDesc[^ ::= ^]
                      ;
```

```
optional_formal_desc:   OptFormalDescNull[@    ::=    "<formal
description>"]
                      | OptFormalDesc [^ ::= ^ ]
                      ;

keywords:   KeyWords    [^ ::= "KEYWORDS" ^]
        ;

informal_desc:  InfDesc [^ ::= "DESCRIPTION" "{" ^ "}"]
             ;

formal_desc:    FormalDesc  [^ ::= "AXIOMS" "{" ^ "}"]
        ;

type_impl:  AdaTyImpl       [^ ::= "IMPLEMENTATION Ada" @]
          | TyImpl          [^   ::=   "IMPLEMEMTATION"   ^   ^
"END"]
          ;

optional_operator_impl: OptOpImplNull   [@ ::= "<operator>"]
                      | OptOpImpl ["OPERATOR" @]
                      ;

operator_impl:  AdaOpImpl   [^ ::= "IMPLEMENTATION Ada" @]
             | PsdlOpImpl   [^ ::= "IMPLEMENTATION" @]
             ;

psdl_impl:  PsdlImpl
         ;

optional_streams:   OptStreamsNull
                  | OptStreams
                  ;

optional_timers:    OptTimersNull
                  | OptTimers
                  ;

optional_control_constraints:OptConConstrNull
            | OptConConstr
            ;

data_flow_diagramList:  DaFlDiagramNil
                      | DaFlDiagramPair
                      ;

data_flow_diagram:  DaFlDiagram
                  ;
```

```
link :          Link
     ;

operator_id:            Id
          ;

streams:        Streams
       ;

type_declList:      TyDeclListNil
             | TyDeclListPair
             ;

type_decl:      TyDecl
         ;

type_name   :   EmptyTyName
            |   TyNameId
            |   TyNameTyId
            ;

timers      :   Timers
            ;

control_constraintsList:ConConstrListNil
            | ConConstrListPair
            ;

constraint  :   Constraint
            ;

optional_predicate :    OptPredicateNull
            | OptPredicate
            ;

optional_id_listList:   OptIdListNil
            | OptIdListPair
            ;

optional_id_list:   OptIdList
            ;

optional_exception_listList:    OptExcepListNil
            | OptExcepListPair
            ;

optional_exception_list:    OptExcepList
            ;
```

```
optional_timer_opList:  OptTiOpListNil
            | OptTiOpListPair
            ;

optional_timer_op:  OptTiOp
            ;


timer_op    :   EmptyTimer_Op
        |   ReadTimer
        |   ResetTimer
        |   StartTimer
        |   StopTimer
        ;

 trigger    :   TriggerNull
            | ByAllTrigger
            | BySomeTrigger
            ;

predicate   :   EmptyPred
            |   NotPred
            |   AndPred
            |   OrPred
            |   Expression
            |   Id
            ;

expression  :   EmptyExpression
            |   ConstExpr
            |   ExprId
            |   TyExpr
            ;

constant    :   EmptyConst
            |   Number
            |   Boolean
            ;

numeric_constant:   EmptyNumber
                |   Real
                |   Integer
                ;

boolean_constant:   EmptyBool
                |   True
                |   False
                ;

expressionList  :   ExpressionListNil
```

APPENDIX E: CONCRETE INPUT SYNTAX

```
/*1*/ TYPE:          TypeLex< "type"|"TYPE" >;
/*2*/ OPERATOR:          OperatorLex<"operator"|"OPERATOR">;
/*3*/ SPECIFICATION:         SpecificationLex
specification"|SPECIFICATION" >
/*4*/ END:           EndLex< "end"|"END" >;
/*5*/ GENERIC:           GenericLex< "generic"|"GENERIC" >;
/*6*/ INPUT:             InputLex< "input"|"INPUT" >;
/*7*/ OUTPUT:            OutputLex< "output"|"OUTPUT" >;
/*8*/ STATES:            StatesLex< "states"|"STATES" >;
/*9*/ INITIALLY:         InitiallyLex< "initially"|"INITIALLY"
>;
/*10*/ EXCEPTIONS:       ExceptionsLex<
"exceptions"|"EXCEPTIONS >;
/*11*/ COMMA:            CommaLex< "," >;
/*12*/ MAXIMUM_EXECUTION_TIME:  MaximumExecutionTimeLex<
"maximum\execution\time"|"MAXIMUM\EXECUTION\TIME >;
/*13*/ MINIMUM_CALLING_PERIOD:  MinimumCallingPeriodLex<
"minimum\calling\period"|"MINIMUM\CALLING\PERIOD" >;
/*14*/ MAXIMUM_RESPONSE_TIME:   MaximumResponseTimeLex<
"maximum\response\time"|"MAXIMUM\RESPONSE\TIME" >;
/*15 */ MICROSEC:        MicrosecLex< "microsec"|"MICROSEC" >;

/*16 */ MS:          MsLex< "ms"|"MS" >;
/*17 */ SEC:             SecLex< "sec"|"SEC" >;
/*18 */ MIN:             MinLex< "min"|"MIN" >;
/*19 */ HOURS:           HoursLex< "hours"|"HOURS" >;
/*20*/ BY_ REQUIREMENTS:     ByRequirementsLex<
"by\requirements"|"BY\REQUIREMENTS" >;
/*21*/ KEYWORDS:         KeywordsLex< "keywords"|"KEYWORDS" >;

/*22*/ DESCRIPTION:      DescriptionLex<
"description"|"DESCRIPTION" >;

/*23*/ TEXT:              TextLex< "text"|"TEXT" >;
/*24*/ AXIOMS:           AxiomsLex< "axioms"|"AXIOMS" >;
/*25*/ IMPLEMENTATION:       ImplementationLex<
"implementation"|"IMPLEMENTATION" >;
/*26 */ ADA:             AdaLex< "ada"|"Ada"|"ADA" >;
/*27*/ GRAPH:            GraphLex< "graph"|"GRAPH" >;
/*28 */PUNCT_PERIOD:     PunctPeriodLex< "." >;
/*29 */ ARROW:           ArrowLex< "->" >;
/*30*/ DATA_STREAM:      DataStreamLex<
"data\stream"|"DATA\STREAM" >;
```

```
/*31 */ COLON:           ColonLex< ":" >;
/*32 */ LEFT_BRACKET:    LeftBracketLex< "[" >;
/*33 */ RIGHT_BRACKET:   RightBracketLex< "]" >;
/*34*/ TIMER:            TimerLex< "timer"|"TIMER" >;
/*35*/ CONTROL_CONSTRAINTS: C o n t r o l C o n s t r a i n t s L e x <
"control\constraints"|"CONTROL\CONSTRAINTS" >
/*36*/ TRIGGERED:        TriggeredLex< "triggered"|"TRIGGERED"
>;
/*37*/ TIME_PERIOD:      TimePeriodLex< "period"|"PERIOD" >;
/*38*/ IF:          IfLex< "if"|"IF" >;
/*39*/ FINISH_WITHIN:       F i n i s h W i t h i n L e x <
"finish\within"|"FINISH\WITHIN" >;
/*40*/ EXCEPTION:        ExceptionLex< "exception"|"EXCEPTION"
>;
/*41*/ RESET:            ResetLex< "reset"|"RESET" >;
/*42*/ START:            StartLex< "start"|"START" >;
/*43*/ STOP:             StopLex< "stop"|"STOP" >;
/*44*/ BY_ALL:           ByAllLex< "by\all"|"BY\ALL" >;
/*45*/ BY_SOME:          BySomeLex< "by\some"|"BY\SOME" >
/*46*/ NOT:         NotLex< "not"|"~"|"NOT" >;
/*47*/ AND:         AndLex< "and"|"&"|"AND" >;
/*48*/ OR:          OrLex< "or"|"|"|"OR" >;
/*49*/ LEFT_PAREN:       LeftParenLex< "(" >;
/*50*/ RIGHT_PAREN:      RightParenLex< ")" >;
/*51*/ TRUE:             TrueLex< "true"|"TRUE" >;
/*52*/ FALSE:            FalseLex< "false"|"FALSE" >;
CLINEBREAK: CLinebreakLex< <NO_WHITESPACE>[\n] >;
LINE:       LineLex< <NO_WHITESPACE>[^\n\}][^\n\}]* >;
LCURLY:     LCurly< [\{] <NO_WHITESPACE> >;
RCURLY:     RCurly< [\}] <INITIAL> >;
WHITESPACE: < [\ \t\n]|([\{][\-][\-][^\}]*[\}]) >;
IDENTIFIER: IdentifierLex< [a-zA-Z][a-zA-Z0-9_\$]* >;
INTEGER:    IntegerLex< [0-9]+ >;
REAL_NUMERAL:   RealLex<  [0-9]+(\.[0-9]+)?([eE][-+]?[0-9]+)?
>;
CHARACTER:  CharacterLex< '.'|'''' >;
STRING:     StringLex< '((')|[^'])((')|[^'])+' >;
```

APPENDIX F: PROTOTYPING EDITOR COMMANDS

The following commands perform the indicated functions:

ESC-^C, exit
^X^C,
^C
return to shell.

^I, execute-command <name>
ESC-^G
specify a command or unique command prefix at COMMAND prompt at the command line.

ESC-^G, illegal-operation
^G,
^X^G
cancel incomplete command key-binding or partial entry on the command line.

ESC-s start-command
execute command with parameters contained in the current form.

ESC-c cancel-command
cancel command awaiting execution.

^X! execute-monitor-command <command-line>
execute UNIX <command-line> put its output into a textfile buffer in a separate window.

ESC-r repeat-command
repeat last command.

^_ return-to-monitor
recursively call shell.

^L redraw-display
refresh screen.

set-parameters
modify editor parameters including indentation, margins, word wrapping, tab stops, and help levels.

ESC-? apropos <keyword>
a listing of commands containing a given keyword.

^X^B list-buffers
a listing of all buffers in a textfile buffer.

^Xb switch-to-buffer <buffer-name>
put <buffer-name> in current window.

new-buffer <buffer-name phylum>
create a new buffer named buffer-name

^X^R read-file <file-name>
replace current buffer contents with file-name, prompt for write of old file in buffer.

^X^V visit-file <file-name>
read a named file into a into a corresponding buffer, replacing the previous contents of the buffer.

^Xs write-current-file
write buffer into its associated file.

^X^W write-named-file <file-name format>
write buffer to file-name in the format specified (text or structure)

^X^M write-modified-files
write each modified file in the buffer to its associated file.

^X^F write-file-exit
write each modified file in the buffer to its associated file then exit the buffer.

^X^I insert-file <file-name>
replace the current selection of the buffer with file-name.

write-selection-to-file <file-name format>
write current selection to file-name in the format specified.

^X2 split-current-window
split the current buffer into two windows each displaying the current buffer but each separately modifiable.

^X1 delete-other-windows
delete all windows except the current one.

^Xd delete-window
delete the current window and replace it with the previous one.

^Xz enlarge-window
raise height of current window by one line.

^X^Z shrink-window
reduce height of current window by one line.

^Xn next-window
go to next window.

^Xp previous-window
go to previous window.

help-off
turn the help pane height to zero.

help-on
turn the help pane height to size set in parameters.

ESC-^Xz enlarge-help
Increase help pane size by one line.

ESC-^X^Z shrink-help
reduce help pane size by one line.

^N forward-preorder
go to the next resting place in preorder, skip over optional constituents. If in text, move down to next line.

^P backward-preorder
go to the previous resting place in preorder, skip over optional constituents. If in text, move up to previous line.

^F right
same as forward-preorder unless text, in which case moves selection one character to the right.

^B left
same as backward-preorder unless text, in which case moves selection one character to the left.

^M forward-with-optionals
go to the next resting place in preorder, stop at optional constituents.

^H backward-with-optionals
go to the previous resting place in preorder, stop at optional constituents.

ESC-^N forward-sibling
skip past all resting places in the current selection, advance to the next preorder sibling in the abstract syntax tree, if applicable, else ascend to the enclosing resting place and advance to its next sibling. Skip over optional constituents.

ESC-^P backward-sibling
skip past all resting places in the current selection, advance to the previous preorder sibling in the abstract syntax tree, if applicable, else ascend to the enclosing resting place and advance to its previous sibling. Skip over optional constituents.

ESC-^M forward-sibling-with-optionals
as forward-sibling, but stopping at optional constituent placeholders.

ESC-^B backward-sibling-with-optionals
as backward-sibling, but stopping at optional constituent placeholders.

ESC-\ ascend-to-parent
go to nearest enclosing resting place.

ESC-< beginning-of-file
go to root of abstract syntax tree.

ESC-> end-of-file
go to extreme right resting place of abstract syntax tree.

advance-after-parse
automatically substituted for forward-with-optionals when following textual entry.

advance-after-transform
automatically substituted for forward-with-optionals when following a transformation command.

forward-after-parse
automatically substituted for forward-preorder when following textual entry.

^A beginning-of-line
got to beginning of line.

^E end-of-line
got to end of line.

scroll-to-bottom
scroll bottom line of window to center of window.

scroll-to top
scroll to put first line of object at top of window.

ESC-! selection-to-top
scroll to put first line of selection at top of window.

^V next-page
move object view one page down.

ESC-v previous-page
move object view one page up.

^Z next-line
move object view one line down.

ESC-z previous-line
move object view one line down.

ESC-{ page-left
move object view one page left.

ESC-} page-right
move object view one page right.

column-left
move object view one column left.

column-right
move object view one column right.

ESC-b pointer-left
move the cursor one character to the left.

ESC-f pointer-right
move the cursor one character to the right.

ESC-p pointer-up
move the cursor one character up.

ESC-d pointer-down
move the cursor one character down.

pointer-long-left
move the cursor eight characters to the left.

pointer-long-right
move the cursor eight characters to the right.

pointer-long-up
move the cursor eight characters up.

pointer-long-down
move the cursor eight characters down.

ESC-, pointer-top-of screen
move cursor to first line of screen.

ESC-. pointer-bottom-of screen
move cursor to last line of screen.

ESC-@ select
select-start followed by select-stop.

select-start
go to the production instance whose unparsing scheme caused the printing of the character located at the cursor and begin dragging.

select-stop
stop dragging.

ESC-t select-transition
toggle between select-start and select-stop.

ESC-(extend
extend-start followed by extend-stop.

extend-start
change selection to the least common ancestor of the apex of the current selection and the production instance which generated the character under the cursor.

extend-stop
stop dragging.

ESC-X extend-transition
toggle between extend-start and extend-stop.

^W cut-to-clipped
move the selection of the current buffer to a buffer named CLIPPED. Replace previous contents of CLIPPED.

ESC-^W copy-to-clipped
copy the selection of the current buffer to a buffer named CLIPPED. Replace previous contents of CLIPPED.

^Y paste-from-clipped
move into the selection of the current buffer from CLIPPED. Contents of CLIPPED is unchanged.

ESC-^Y copy-from-clipped
copy into the selection (a placeholder) of the current buffer from CLIPPED. Contents of CLIPPED is unchanged.

ESC-^T copy-text-from-clipped
copy text into a text buffer immediately preceding the selection of the current buffer from CLIPPED. Contents of CLIPPED is unchanged.

^K delete-selection
move into the DELETE buffer the selection of the current buffer. Replace previous contents of DELETE.

^D delete-next-character
delete character under cursor.

DEL delete-previous-character
delete character to left of cursor.

ESC-d erase-to-end-of-line
erase from character under cursor to end of line.

ESC-DEL erase-to-beginning-of-line
erase from character before cursor to beginning of line.

^K delete-selection
delete entire line.

^J new-line
add a line in the text buffer.

text-capture
put the text of the current selection into a text buffer.

^X^U undo
restore the selection to its state before the text-capture and delete the text buffer.

dump-on
split the window into the current buffer and a "dump buffer". Display the attributes of the corresponding apex of the selection and the attributes of the non-resting-place nodes immediately below the apex in the "dump buffer".

dump-off
turn off the dynamic updating of the "dump buffer".

show-attribute <attribute-name buffer-name>
copy the value of attribute-name of the current selection into buffer-name. This gives buffer-name the syntactic mode of the attribute.

write-attribute <attribute-name file-name>
write attribute-name, the attribute of the current selection into file, file-name.

ESC-^F search-forward <text phylum name operator-name>
forward preorder search from current selection to next occurrence of a STR value corresponding to text, an instance of a term of the given phylum, or an instance of a term of the given phylum, or an instance of a term having the given operator. After finding the end of the object, continue search by wrapping around to the root.

ESC-^R search-reverse <text phylum-name operator-name>
reverse preorder search from current selection to next occurrence of a STR value corresponding to text, an instance of a term of the given phylum, or an instance of a term of the given phylum, or an instance of a term having the given operator. After finding the end of the object, continue search by wrapping around to the rightmost leaf. [Ref. 10]

The mouse can be used to point anywhere on the object pane, and click to a new selection. The mouse can be used in conjucntion with the select-start/select-stop commands to drag between characters in a given production. [Ref. 10]

The middle button activates a menu with the following commands: edit, cursor, windows, file, and search. [Ref. 10]

The edit command, on dragging the mouse to the right expands to the following commands :

```
apropos
text-capture
undo
cut-to-clipped
copy-to-clipped
paste-from-clipped
copy-from-clipped
copy-text-from-clipped
delete-selection
repeat-command
alternate-unparsing-toggle
alternate-unparsing-on
alternate-unparsing-off
set-parameters
dump-on
dump-off.  [Ref.10]
```

The cursor command, on dragging the mouse to the right expands to the following commands :

```
ascend to parent
forward-preorder
forward-sibling
forward-sibling-with-optionals
forward-with-optionals
backward-preorder
backward-sibling
backward-sibling-with-optionals
backward-with-optionals
end-of-file
selection-to-top.  [Ref.10]
```

The windows command, on dragging the mouse to the right expands to the following commands :

```
split-current-windows
delete-other-windows
delete-window
help-off
help-on
```

enlarge-help
shrink-help. [Ref.10]

The windows command, on dragging the mouse to the right expands to the following commands :

list-buffers
switch=to-buffer
new-buffer
read-file
visit-file
insert-file
write-current-file
write-named-file
write-modified-file
write-file-exit
write-selection-to-file
write-attribute
exit. [Ref.10]

The search command, on dragging the mouse to the right expands to the following commands :

search-forward
search-backward. [Ref.10]

Clicking on the right mouse button gives the transformation block which expands according to the context of the highlighted item. [Ref. 10]

APPENDIX F: PARTIAL GLOSSARY OF TERMS FROM THE SYNTHESIZER GENERATOR REFERENCE MANUAL

abstract syntax phylum - in an attribute equation of an entry declaration, the abstract syntax phylum is a variable denoting the selected subterm or sublist of the edited buffer ([Ref. 10] page 32)

abstract syntax - language meaning as defined by grammar

alphabet - a finite set of symbols

alternate unparsing scheme -
```
***operator [ left-side : right-side ];
operator [ left-side ::= right-side ]; ****
```

attribute equation - (associated with productions of the abstract syntax) defines translation from text to abstract syntax

- defines an attribute as the value of an expression defined in terms of other attributes of the production and is associated with every production in the grammar([Ref. 10] page 23).

```
************ phylum : operator , ... , operator { equations }
             | operator , ... , operator { equations }
             ...
             | operator, ... , operator {equations }
             ;   ([Ref. 10] page 26)
```

attribute - used to describe context dependent features of a language attached to a phylum by a declaration that specifies the name of the phylum, the type of each of the attributes and whether each attribute is synthesized or inherited.([Ref. 10] page 23) An attribute's type may be one of the built-in phyla or user defined.

- characteristic, specific thing which means something within the context of the language or what one is doing with the language (i.e., data type of integer, or coercion of an integer to a real type)

- synthesized
    - ******phylum0{
      .synthesized phylum1 attribute1;
      ...
      inherited phylumk attributek;
      }; **********

    Within the attribute equations of a production, each synthesized and inherited attribute of a phylum occurring in the production is a variable whose value is determined by its attribute equation in the production. ([Ref. 10] page 32)

- inherited

- local
    Used to attach attributes to productions rather than phyla. Within the attribute equations of a production or entry declaration, each local attribute of the production or entry declaration in a variable whose value is determined by its corresponding attribute equation. A local attribute's name denotes that attribute; the attribute's declaration must preceed any use of it's name in an attribute equation.

attribute expression - used to force the attribution of a previously unattributed term:

**********expression {equations}.attribute**********([Ref. 10] page 47)
- the value of this expression is computed as follows:
(a) the expression is evaluated, yielding some attribute (but as yet unattributed) term T of a phylum for which attribute is an attribute,
(b) the inherited attributes of T are defined by the given equations
(c) the value of T.attribute is computed by demand and is returned. ([Ref. 10] page 47)

attribute grammar - extends a context free grammar by attaching attributes to the symbols to the symbols of the grammar

attribute types - either built-in phyla or user defined phyla, using exactly the same rules as in the definition of abstract syntax.

- defined with grammar rules ([Ref. 10] page 47)

attribute declaration - associate attributes with nonterminals and productions

attributed - decorated with computed values that characterize the term

attribution rules- specify syntax directed computations on terms

buffer selection - either an entire subtree of the buffer contents or an interior subtree that denotes a sublist of a list phylum.

- displayed according to the unparsing schemes of its constituent productions i.e., either editable as text or immutable (an immutable section may have constituents that are editable)

- if a selection is editable as text, it can be modified character-by-character ([Ref. 10] page 61). When the buffer selection is changed, the portion just edited is parsed and syntactacally checked and if found to be in error, the error is marked and the next selection is cancelled.

buffer - contain objects

- normally a file is contained in a buffer

circularity - ([Ref. 10] page 29)

- type 1 - indicates a circularity in the dependencies of an individual production

- type 2 - indicates a circularity in the approximation of the production's transitive dependencies

- type 3 - circularity induced by the dependencies that are added between attribute partitions.

clipping mode - w is the absolute right margin ([Ref. 10] page 55)

combination attribute declaration and associated production -

```
********phylum : operator, ... , operator (argument phyla)
{equations}
        |  operator,  ...  ,  operator  (argument  phyla)
{equations}          ...
         operator,   ...   ,   operator   (argument   phyla)
{equations}          ;
```

completing term - a distinguished term that completes a phylum declaration ...(cannot be circularly defined)

concrete input syntax - determines the structure of edited text and its translation to abstract form.

concrete syntax phylum - a variable denoting the parse tree of the parsed input text.

concrete input syntax - how the terms of a phylum can be given a concrete input representation so that text files can be read and components of a term can be entered as text. ([Ref. 10] page 61)

conditional unparsing item - a list of unparsing items enclosed in square brackets. (example [Ref. 10] page 60)

conditional expression -

- with expression - a multi-branch permitting discrimination based on the structure of the value of a given expression.

```
********with (expression)(
        pattern1: expression1,
        pattern2: expression2,
        ...
        patternn: expressionn
        )*******************
 conditional   expression   -   ****expression1   ?
expression2:expression3 *** ([Ref. 10] page 44)
```

consistently attributed derivation tree - as edit, transform from one syntactically well formed tree to another

context-free grammar - (basically means that can determine the meaning of a language without having to have it in context)

- finite set of variables (nonterminals) each of which represents a language
- if meaning or a string is unambiguous using a PDA (push down automaton)
- used recursive descent parsing or a table driven scheme.

declaration - used to specify an editor

demand attribute - an attribute that will be given a value only when necessary i.e., only when a demand is placed on them for their value rather than being automatically maintained whether their values are needed of not. A demand for attribute a arises either directly from a need to display a on the screen or indirectly from the need to evaluate another attribute that depends on a. Demand attributes may be arguments of regular attribute and vice versa.

********demand synthesized phylum attribute ;
demand inherited phylum attribute ;
demand local phylum attribute ; **********

derivation tree - superimpose a structure on the words of a language that is useful in applications such as the compilation of programming languages

- also represents expressions and control structures - a tree that defines how a tree can be parsed.

display representation of term t - ([Ref. 10] page 52)
- display string S(t)
- display array A(S(t),w)

display string - let t be a term. The string S(t) is defined inductively : ([Ref. 10] page 53)

- If t is a value of a primitive phylum, then S(t) is chosen according to the table of display formats found on [Ref. 10] page 53

- If t is not primitive, it is op((t1, ..., tn) for some operator op and subterms t1, ...tn.

display array - A(S(t), w), the two dimensional textual representation of a term t is created by interpretint each character of the display string during a left to right scan. ([Ref. 10] page 54)

editing transaction - text editing, structural transformation, or system command

entry declaration - in order for a selection with apex operator op of phylum p to be text editable, there must exist at least one entry declaration for p.

```
* * * * * * * * * abstract-syntax-phylum ~
concrete-syntax-phylum.attribute ;
 abstract-syntax-phylum~concrete-syntax
-phylum.attribute{equations};
********************************
```
([Ref. 10] page 63)

entry declaration - defines what part (subset) of the input syntax is recognized at any part of the program (dependent on editor cursor position).

- defines the correspondence between the abstract syntax of editable objects and subsets of the input syntax

- used to temporarily establish a linkage between a nonterminal in an abstract syntax tree being edited and the root nonterminal of a parse tree for some analysed text. ([Ref. 10] page 32)

equation declaration - defines the value of attributes in terms of other attributes that occur in the production

evaluation scheme -
ordered - (Default) a condition sufficient to guarantee noncircular grammar ([Ref. 10] page 28)
nonordered - (not recommended) [Ref. 10] page 30

expression - a formula denoting an unattributed term. Used in attribute equations of productions, function

declarations, transformation declarations and unparsing declarations. ([Ref. 10] page 31)

factoring - a process used in both production and lexeme declaration to describe the rule using less space ([Ref. 10] pages 12 & 15)

formal parameter - Each formal parameter of a function is a variable that denotes the value of the corresponding actual parameter passed to the function. Formal parameters are named by identifiers. ([Ref. 10] page 34)

format_strings - see interpretation modes

function declaration - operation to abstract something into a function

**phylum0 identifier0 (phylum1 identifier1, ... , phylumk identifierk)
{expression}; ********** ([Ref. 10] page 30)
- declares identifier0 to be a k-ary function with result phylum phylum0, and has, for each 1 <= i <= k, a formal parameter name
identifieri of type phylumi. The body of the function is an expression over identifier1, ... , identifierk that must evaluate to a term in the result phylum, phylum0.([Ref. 10] page 30)

function restrictions
- argument type phyla must be declared
- declarations are global
- can't be declared within another function
- can't access attribute values not passed as parameters
- functions are second class citizens
- functions can be recursive
- details on [Ref. 10] page 30

function - a rule for determining a new term given k argument terms

- declarations of functions (written in SSL)
- predefined library routines (sec 2.5 [Ref. 10])
- foreign functions (written in C) (sec 5.1 [Ref. 10])

grammar - a type definition mechanism that in which the nonterminal symbols are type names

grammar rules - productions of a context-free grammar

graph - consists of a finite set of vertices and a set of pairs of vertices called edges

identifier - a sequence of letters, digits, or underscores that are used in editor specifications to name phyla, operators, functions, and attributes.

identifier convention - lower_case - phyla of abstract syntax

CapitalizedWords - operators and functions
lower_case - attributes
CAPITALIZED - phyla of lexemes and phyla used for attribute types
CapitalizedWords - phyla for input syntax

indentation commands - ([Ref. 10] pages 55 & 56)

%t - moves the left-margin one indentation unit to the right

%b - moves the left-margin one indentation unit to the left

%n - displays text following it on the next line

%o - optional line breaks (prefered places where the display of a line may be divided into separate lines)

%c - connected break (permits specification of a display in which either all text of an unparsing grouping appears on the same line or all connected line breaks in the group are taken.-SEE ALSO table [Ref. 10] page 56

input symbol - next token under pointer in program

input syntax - determines the structure of edited text and its translation into abstract form

insert-file - Let s be either the result of text editing or

the contents of a text file read in by the insert-file command. Then, s is translated to a term as following

1. The string is parsed and the parse tree t' of phylum p' is produced.
2. the parse tree t' is attributed and some designated attribute a of p' provides the value t that replaces the currently selected subterm. After the attribute p'.a is extracted from t', the parse tree is thrown away.([Ref. 10] page 62)

interpretation modes - based on Synthesizer Specification Language (SSL) compile-time flag ''formal_strings'' ([Ref. 10] page 54)

- if format_strings is true - every percent sign that occurs in S(t) will be interpreted as a formatting command

- if format_strings is false - percent signs are interpreted as formatting commands only if they arise from quoted unparsing items of unparsing schemes (default mode)

k-ary operator - a constructor-function mapping k terms to a term.

language - set of strings of symbols from some one alphabet

let-expression - bind values to names or when the structure of a value is known in part and one desires local names for its constituents.([Ref. 10] page 45)

******* let pattern1 = expression1 in (expression) *****

lexemes - phyla consisting of sets of strings - sub-phyla of STR (predefined primative phylum denoting all possible character strings)

*******phylum : lexeme name < regular expression>*********
which means that if you can generate a string using the regular expression, then that string belongs to the named phylum (Ref. 10] pages 13,14)

- psdl.lex.ssl is an example of lexemes

matched value - in a with expression, the first pattern that matches the expression value([Ref. 10] page 42).

modification - a change to a program by transformation, command or text editing

noncircularity - it must not be possible to build a derivation tree in which attributes are defined circularly.

nonterminal - a symbol, which when applied to an input symbol, points to a new production

nullary operator - the result of applying itself

object- terms that are created, modified and/or destroyed

operator - (is typed)- a named production- used in computational expressions both as a record constructor and as a selector that discriminates between variants

output attribute - a variable ([Ref. 10] page 25)

parenthesized for grouping - (expression) ([Ref. 10] page 46)

parsing declaration - define productions of a grammar to be used for parsing text and the phylum of corresponding parse trees

- to translate a string s, create a parse tree t' of some given phylum p'. Associate concrete input languages with phyla in such a way that each string uniquely determines a parse tree. ([Ref. 10] page 64)

path - a sequence of vertices such that there is an edge connecting every vertice between the origin and the destination.

pattern - <phylum> and [phylum] are patterns

- pattern variables are patterns
- both ''*'' and ''default'' are patterns
- a k-ary operator op applied to k patterns is a pattern and is written ********** op(pattern1,...patternk)*********
- a pattern variable followed by the keyword ''as'' followed by a pattern is a pattern.

********** pattern-variable as pattern *****

- Let p be a pattern and t be a term. Then p is said to match if :
- when p is [phylum] and t is equal to the phylum's completing term or when p is <phylum> and t is equal to the phylum's placeholder term
- when p is the binding occurrence of a pattern variable pv, in which case pv is bound to t. when p is a bound occurence of a pattern variable pv that has been bound to some term t' and t==t'
- when p is either ''*'' or ''default''
- when p is op(p1, ... , pk) and t is op(t1, ..., tk), and pi matches ti for all i between 1 and k. If the op is nullary and has been declared earlier in the specification, the parentheses may be omitted.
- When p is pv as p' and p' matches t and either (e1) this is the defining occurence of pv or (e2) it is a bound occurence of pv and pv has been bound to some pattern values - variables bound to the matched constituents in a with statement ([Ref. 10] page 42)

pattern variables - can inherit attributes ([Ref. 10] page 34)

phylum - the set of terms derived from a given nonterminal symbol

- contains a distinguished term called a completing term
- contains a distinguished term called a placeholder term
- a nonterminal symbol, taken as a type name denoting a set of values
- collections of abstract terms that can be used to compute with, but have no concrete representation.

phylum declaration - defines productions, nonterminal symbols and operator names - two kinds:

production - defines a new operator and includes all terms constructible from by that operator in a given phylum.

phylum1 : operator(phylum1 phylum2 ... phylumk);

lexeme - phyla consisting sets of strings; a sub-phyla of STR

phylum mixing production and lexeme - see [Ref. 10] page 16

phylum occurence - Within the attribute equations of a production each phylum occurence is a variable whose value is the term derived from that occurence in the particular instance of that production. Let X be one of the phyla occurring in the given production. If phylum X occurs only once in the production, then the name X is sufficient to identify that occurrence. If phylum X occurs more than once, then the names X$1,X$@, etc. identify the different occurrences of X in the production. The abbreviation $$ is a synonym for phylum0. ([Ref. 10] page 32)

phylum types - primitive

- primitive values are considered to be derivation trees derived from an infinite collection of productions associated with the predefined nonterminals(BOOL, INT, REAL, CHAR, etc.)

- user defined
  - declare in terms of other phyla or recursively in terms of themselves.
  - list
  - a linear list of phyla
  - optional
  - a phyla whose placeholder is not required to be filled in
  - an optional phylum can have any number of productions, but one of them must be a nullary production; the completing term is the term constructed from this nullary term
  - textfile

- predefined as a list of zero or more lines of text
- root declaration
- declares the phylum of editable objects.

placeholder term - a distinguished term that ...when selected, can be replaced with an edited object; conversely, when an object is deleted, it is replaced with the placeholder term

- when unparsed, represents a class of objects that can be inserted at that location - in the case of a non-optional non-list phylum, the placeholder term is the completing term - ****<phylum>*********([Ref. 10] page 35)

precedence -
- all unary operators have a higher precedence than binary operators

- binary operator precedence ([Ref. 10] page 47 table)

precedence declaration - define precedence and associativity of terminal symbols - ([Ref. 10] page 47)

primitive phyla (predefined) -
BOOL - Truth values
INT - Integers
REAL - Floating point
CHAR - Characters
STR - Char strings
PTR - References to SSL values
ATTR -References to attributes
TABLE - Hash tables

principal unparsing scheme - used by default when the production is instantiated. After that, the user may choose between principle and alternate schemes.
********phylum : operator [unparsing scheme][unparsing scheme]; ******** (p49)

production - a phylum declaration that defines a new operator and includes all terms constructible by that operator in a given phylum.

production rules - direct the translator to the next potential language construct(s)}

property declarations - specify special properties of phyla i.e., - optional

- a phylum represented by a null phylum(empty string completing term) that can be replaced or ignored
  *****optional phylum1, ... , phylumk; ************

- an optional phylum can have any number of productions, but one of them must be a nullary production; the completing term is the term constructed from this nullary term

- lists - a list phylum is treated like a linear list of items; an item can be inserted before, after, or in the middle of the list ([Ref. 10] page 40 lists built-in operators :: (concatenation), @ (append))
  *****list phylum1, ... , phylumk; ************
  since all insertions are made at placeholders, in a list, a placeholder may appear before and after every item. For both the list and optional list phyla, the ''in between'' placeholders are implicitly optional, and have the properties of optional phyla. - A list phylum must have exactly two productions([Ref. 10] page 21) a nullary production and a binary production that is right recursive
  ****************list calc;

```
    calc: CalcNil()
     |CalcPair(exp calc)

    ;   ***************************
```

- the completing term of a nonoptional list phylum is the single element list constructed by applying the binary production {|CalcPair(exp calc)} to the completing term of its left son {the completing term of exp : Null() } and to the list's nullary term {CalcNil() } to construct the expression {CalcPair(Null(), CalcNil())}

- the binary production of calc applied to the completing term of exp and the nullary production of calc.
  ********************[phylum]*********([Ref. 10] page 35)

- optional list
  *****optional list phylum1, ... , phylumk; ************
  The difference between a list phylum and an optional list phylum is that the list phylum treats lists of 1 or more (the placeholder term remains until at least one item has

been inserted into the list) while the optional list phylum treats lists of 0 or more.

- the completing term of an optional list phylum is the constant term constructed from the nullary production. the place holder term is the same as a nonoptional list phylum ([Ref. 10] page 22).

relational operations - a total order is defined on the universe of values ([Ref. 10] page 41)

resting place - the node at which the apex of a selection can rest (such nodes are selectaable)

- a place where the editor stops to insert a phylum.

resting-place-denoters - symbols (@, ^, ..) that correspond to the phyla occurences in the production whose display format is being defined.([Ref. 10] page 51)

- determine the resting places of a term, those nodes at which the apex of a selection can rest. Such nodes are said to be selectable. ([Ref. 10] page 51)

root declaration - declares the phylum of editable objects.- specifies the root nonterminal symbol of the grammar-

scanner - program that feeds tokens to the parser (tokenizes the input file for the parser)

selectable - those nodes at which the apex of a selection can rest. Such nodes are said to be selectable. ([Ref. 10] page 51)

set-parameters - command to set tab length, margins etc. ([Ref. 10] page 54) & sec 3.1

term - the application of a k-ary operator to k terms of the appropriate phyla

- the objects created, modified and and destroyed by the editor user

- values computed as attributes of terms are themselves terms

- derivation tree derived from nonterminal symbol

- used as both abstract representations of objects to be edited and as computational values

- derivation tree with respect to the underlying abstract syntax of the language

- a given term, in different instances may be both an attribute value and a piece of the abstract syntax of an object being edited.

terminal symbols - a type definition mechanism in which nonterminal symbols are type names and each nonterminal symbol, taken as a type name, denotes a set of values known as a phylum.

terminal - primitive symbols in a language

text-capture - makes the textual representation of the {apex?} term available for modification in an edit buffer. Any overwriting of or erasing of a character implies a text capture ([Ref. 10] page 49-50)

textfile - a phylum that is predefined as a list of zero or more lines of text - User specified phyla can incorporate textfile constituents ([Ref. 10] page 22).

```
*****   list textfile;
 textfile: TextFileNil()
 |  TextFilePair( textline textfile)
 ;
 textline: TextLineNil()
 |  TextLine(STR)
 ;          **************
```

token - defined by rules (not in grammar) in the lexigraphical rules

transformation - determines a replacement value for the selected subterm as a function of it's current value

transformation declaration - specifies how to restructure an object when the component where the cursor is pointing matches a given structural pattern

tree - directed graph such that there is one root and every child except the root has only one parent, and successors from each vertex are ordered from left to right.

unparsing declaration - define the output representation of a phylum

unparsing declarations - how the terms of a phylum can be given a concrete output representation so that they can be displayed on the screen or written to a text file.

unparsing declarations - ([Ref. 10] page 48)
- two dimensional textual representation of terms
- selectable components of terms
- default editing modes of the selectable components

*******phylum : operator [unparsing-scheme]; *********
which specifies unparsing properties for the production:

```
phylum : operator (arguments) ;
```

unparsing items - interspersed among the resting-place-denoters on the right hand side of an unparsing scheme are zero or more unparsing items. ([Ref. 10] page 52)

- quoted STR constants
- phylum occurences of the corresponding production
- attributes of phylum occurences of the given production
- conditional unparsing items

unparsing lists - need to separate rather than terminate list items ([Ref. 10] page 59)

upward remote attribute sets - within the equations of a production p, it is possible to refer nonlocally to the attributes of a different production that necessarily

occurs above instance of production p in a term. ([Ref. 10] page 33 for details)

value of the variable - The term bound to a variable.

variable - a name bound to a term. The term bound to the variable is known as the value of the variable.([Ref. 10] page 31)

variable restrictions
- can't be rebound
- if involve an attribute, then the attribute must be declared before before the variable ([Ref. 10] page 31)

well-formedness - there must be exactly one equation for every output attribute of each production.

word-wrapping mode - w is the minimum of the absolute right margin and the current width of the window in which the term is being displayed. ([Ref. 10] page 55)

LIST OF REFERENCES

1. Berzins, V., "Software Engineering," class notes provided at Naval Postgraduate School, Monterey, California, Spring Quarter 1988.

2. MacLennan, B.J., Principles of Programming Languages: Design, Evaluation, and Implementation, 2d ed, pp. 261-263, 328, Holt, Rinehart and Winston, 1987.

3. Luqi, Rapid Prototyping for Large Software System Design, Ph.D. Thesis, University of Minnesota, Duluth, Minnesota, May 1986.

4. Reps, T.W., and Teitelbaum, T., The Synthesizer Generator, pp. 18-67, Springer-Verlag, 1988.

5. Knuth, D.E., "Semantics of Context-Free Languages," Mathematical Systems Theory, v.2, pp. 127-145, June 1968.

6. Knuth, D.E., "Semantics of Context-Free Languages: Correction," Mathematical Systems Theory, v.5, pp. 95-96, March 1971.

7. Reps, T.W., and Teitelbaum, T., The Synthesizer Generator Reference Manual, 2d ed., pp. 1-105, Department of Computer Science, Cornell University, 1987.

8. Notkin, D., "The GANDALF Project," The Journal of Systems and Software, v.5, pp. 91-106, May 1985.

9. Ellison, R.J., and Staudt, B.J., "The Evolution of the GANDALF System," The Journal of Systems and Software, v.5, pp. 107-121, May 1985.

10. Luqi, Berzins V., and Yeh, R.T., "A Prototyping Language for Real-Time Software," IEEE Transactions on Software Engineering, v. 14, No. 10, October 1988.

11. Reps, T.W., Generating Language-Based Environments, Ph.D. Thesis, Cornell University, Ithaca, New York, August 1982.

12. Luqi and Ketabchi, M. "A Computer-Aided Prototyping System," IEEE Software, pp. 66-72, March 1988.

13. Galik, D., A Conceptual Design of a Software Base Management System for the Computer Aided Prototyping System, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

14. Thorstenson, R., A Graphical Editor for the Computer Aided Prototyping System (CAPS), Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

15. Raum, H., The Design and Implementation of an Expert User Interface for the Computer Aided Prototyping System, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

16. Marlowe, L., A Scheduler for Critical Timing Constraints, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

17. Altizer, C., Implementation of a language Translator for a Computer Aided Rapid Prototyping System, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

18. Wood, M., Run-Time Support for Rapid Prototyping, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, Virginia 22304-6145

2. Library, Code 0142 2
   Naval Postgraduate School
   Monterey, California 93943-5002

3. Office of Naval Research 1
   Office of the Chief of Naval Research
   Attn. CDR Michael Gehl, Code 1224
   800 N. Quincy Street
   Arlington, Virginia 22217-5000

4. Space and Naval Warfare Systems Command 1
   Attn. Dr. Knudsen, Code PD 50
   Washington, D.C. 20363-5100

5. Ada Joint Program Office 1
   OUSDRE(R&AT)
   Pentagon
   Washington, D.C. 20301

6. Naval Sea Systems Command 1
   Attn. CAPT Joel Crandall
   National Center #2, Suite 7N06
   Washington, D.C. 22202

7. Office of the Secretary of Defense 1
   Attn. CDR Barber
   STARS Program Office
   Washington, D.C. 20301

8. Office of the Secretary of Defense 1
   Attn. Mr. Joel Trimble
   STARS Program Office
   Washington, D.C. 20301

9. Commanding Officer 1
   Naval Research Laboratory
   Code 5150
   Attn. Dr. Elizabeth Wald
   Washington, D.C. 20375-5000

10. Navy Ocean System Center 1
Attn. Linwood Sutton, Code 423
San Diego, California 92152-5000

11. National Science Foundation 1
Attn. Dr. William Wulf
Washington, D.C. 20550

12. National Science Foundation 1
Division of Computer and Computation Research
Attn. Dr. Peter Freeman
Washington, D.C. 20550

13. National Science Foundation 1
Director PYI Program
Attn. Dr. C. Tan
Washington, D.C. 20550

14. Office of Naval Research 1
Computer Science Division, Code 1133
Attn. Dr. Van Tilborg
800 N. Quincy street
Arlington, Virginia 22217-5000

15. Office of Naval Research 1
Applied Mathematics and Computer Science, Code 1211
Attn. Mr. J. Smith
800 N. Quincy street
Arlington, Virginia 22217-5000

16. Defense Advanced Research Projects Agency (DARPA) 1
Integrated Strategic Technology Office (ISTO)
Attn. Dr. Jacob Schwartz
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

17. Defense Advanced Research Projects Agency (DARPA) 1
Integrated Strategic Technology Office (ISTO)
Attn. Dr. Squires
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

18. Defense Advanced Research Projects Agency (DARPA) 1
Integrated Strategic Technology Office (ISTO)
Attn. MAJ Mark Pullen, USAF
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

19. Defense Advanced Research Projects Agency (DARPA) 1
Director, Naval Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

20. Defense Advanced Research Projects Agency (DARPA) 1
Director, Strategic Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

21. Defense Advanced Research Projects Agency (DARPA) 1
Director, Prototype Projects Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

22. Defense Advanced Research Projects Agency (DARPA) 1
Director, Tactical Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

23. COL C. Cox, USAF 1
JCS (J-8)
Nuclear Force Analysis Division
Pentagon
Washington, D.C. 20318-8000

24. LTCOL Kirk Lewis, USA 1
JCS (J-8)
Nuclear Force Analysis Division
Pentagon
Washington, D.C. 20318-8000

25. U.S. Air Force Systems Command 1
Rome Air Development Center
RADC/COE
Attn. Mr. Samuel A. DiNitto, Jr.
Griffis Air Force Base, New York 13441-5700

26. U.S. Air Force Systems Command 1
Rome Air Development Center
RADC/COE
Attn. Mr. William E. Rzepka
Griffis Air Force Base, New York 13441-5700

27. Professor Luqi 1
Code 52LQ
Naval Postgraduate School
Computer Science Department
Monterey, California 93943-5100

28. LT Scott W. Porter, USN 1
Puget Sound Naval Shipyard (PSNS)
Bremerton, Washington 98314-5000

Thesis
P7444 Porter
c.1 Design of a syntax
directed editor for
PSDL.

Thesis
P7444 Porter
c.1 Design of a syntax
directed editor for
PSDL.

Design of a syntax directed editor for P

3 2768 000 81346 3
DUDLEY KNOX LIBRARY